

Reinhard Wilhelm (Ed.)

LNCs 2027

Compiler Construction

10th International Conference, CC 2001
Held as Part of the Joint European Conferences
on Theory and Practice of Software, ETAPS 2001
Genova, Italy, April 2001, Proceedings



Springer

Lecture Notes in Computer Science

Edited by G. Goos, J. Hartmanis and J. van Leeuwen

2027

Springer

Berlin

Heidelberg

New York

Barcelona

Hong Kong

London

Milan

Paris

Singapore

Tokyo

Reinhard Wilhelm (Ed.)

Compiler Construction

10th International Conference, CC 2001

Held as Part of the Joint European Conferences
on Theory and Practice of Software, ETAPS 2001
Genova, Italy, April 2-6, 2001

Proceedings



Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editor

Reinhard Wilhelm
Universität des Saarlandes, Fachrichtung Informatik
Postfach 15 11 50, 66041 Saarbrücken, Germany
E-mail: wilhelm@cs.uni-sb.de

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Compiler construction : 10th international conference ; proceedings /
CC 2001, held as part of the Joint European Conferences on Theory and
Practice of Software, ETAPS 2001, Genova, Italy, April 2 - 6, 2001.
Reinhard Wilhelm (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ;
Hong Kong ; London ; Milan ; Paris ; Singapore ; Tokyo : Springer, 2001
(Lecture notes in computer science ; Vol. 2027)
ISBN 3-540-41861-X

CR Subject Classification (1998): D.3.4, D.3.1, F.4.2, D.2.6, I.2.2, F.3

ISSN 0302-9743

ISBN 3-540-41861-X Springer-Verlag Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer-Verlag. Violations are liable for prosecution under the German Copyright Law.

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH

<http://www.springer.de>

© Springer-Verlag Berlin Heidelberg 2001
Printed in Germany

Typesetting: Camera-ready by author, data conversion by PTP-Berlin, Stefan Sossna
Printed on acid-free paper SPIN: 10782418 06/3142 5 4 3 2 1 0

Foreword

ETAPS 2001 was the fourth instance of the European Joint Conferences on Theory and Practice of Software. ETAPS is an annual federated conference that was established in 1998 by combining a number of existing and new conferences. This year it comprised five conferences (FOSSACS, FASE, ESOP, CC, TACAS), ten satellite workshops (CMCS, ETI Day, JOSES, LDTA, MMAABS, PFM, RelMiS, UNIGRA, WADT, WTUML), seven invited lectures, a debate, and ten tutorials.

The events that comprise ETAPS address various aspects of the system development process, including specification, design, implementation, analysis, and improvement. The languages, methodologies, and tools which support these activities are all well within its scope. Different blends of theory and practice are represented, with an inclination towards theory with a practical motivation on one hand and soundly-based practice on the other. Many of the issues involved in software design apply to systems in general, including hardware systems, and the emphasis on software is not intended to be exclusive.

ETAPS is a loose confederation in which each event retains its own identity, with a separate program committee and independent proceedings. Its format is open-ended, allowing it to grow and evolve as time goes by. Contributed talks and system demonstrations are in synchronized parallel sessions, with invited lectures in plenary sessions. Two of the invited lectures are reserved for “unifying” talks on topics of interest to the whole range of ETAPS attendees. The aim of cramming all this activity into a single one-week meeting is to create a strong magnet for academic and industrial researchers working on topics within its scope, giving them the opportunity to learn about research in related areas, and thereby to foster new and existing links between work in areas that were formerly addressed in separate meetings.

ETAPS 2001 was hosted by the Dipartimento di Informatica e Scienze dell’Informazione (DISI) of the Università di Genova and was organized by the following team:

Egidio Astesiano (General Chair)
Eugenio Moggi (Organization Chair)
Maura Cerioli (Satellite Events Chair)
Gianna Reggio (Publicity Chair)
Davide Ancona
Giorgio Delzanno
Maurizio Martelli

with the assistance of Convention Bureau Genova. Tutorials were organized by Bernhard Rumpe (TU München). Overall planning for ETAPS conferences is the responsibility of the ETAPS Steering Committee, whose current membership is:

Egidio Astesiano (Genova), Ed Brinksma (Enschede), Pierpaolo Degano (Pisa), Hartmut Ehrig (Berlin), José Fiadeiro (Lisbon), Marie-Claude Gaudel (Paris), Susanne Graf (Grenoble), Furio Honsell (Udine), Nigel

Horspool (Victoria), Heinrich Hußmann (Dresden), Paul Klint (Amsterdam), Daniel Le Métayer (Rennes), Tom Maibaum (London), Tiziana Margaria (Dortmund), Ugo Montanari (Pisa), Mogens Nielsen (Aarhus), Hanne Riis Nielson (Aarhus), Fernando Orejas (Barcelona), Andreas Podelski (Saarbrücken), David Sands (Göteborg), Don Sannella (Edinburgh), Perdita Stevens (Edinburgh), Jerzy Tiuryn (Warsaw), David Watt (Glasgow), Herbert Weber (Berlin), Reinhard Wilhelm (Saarbrücken)

ETAPS 2001 was organized in cooperation with

the Association for Computing Machinery
the European Association for Programming Languages and Systems
the European Association of Software Science and Technology
the European Association for Theoretical Computer Science

and received generous sponsorship from:

ELSAG
Fondazione Cassa di Risparmio di Genova e Imperia
INDAM - Gruppo Nazionale per l'Informatica Matematica (GNIM)
Marconi
Microsoft Research
Telecom Italia
TXT e-solutions
Università di Genova

I would like to express my sincere gratitude to all of these people and organizations, the program committee chairs and PC members of the ETAPS conferences, the organizers of the satellite events, the speakers themselves, and finally Springer-Verlag for agreeing to publish the ETAPS proceedings.

January 2001

Donald Sannella
ETAPS Steering Committee chairman

Preface

The International Conference on Compiler Construction (CC) is a forum for the presentation and discussion of recent developments in programming language implementation. It emphasizes practical methods and tools. CC 2001 was the tenth conference in the series.

The CC conference originated as a series of workshops started by Günter Riedewald in East Germany in 1986. In 1992 the series was relaunched by Uwe Kastens in Paderborn. In 1994 CC joined ESOP and CAAP in Edinburgh as it did 1996 in Linköping. CC federated with ESOP, FOSSACS, and TACAS to form ETAPS in 1998 and became annual. The number of submissions has shown a nice increase. The program committee received 69 submissions for CC 2001, from which 22 high-quality papers were selected for presentation. These papers are included in these proceedings. The areas of program analysis and architecture received the highest number of submissions and were rewarded with the highest number of accepted papers. Exploiting the intra-processor parallelism and improving the locality of memory referencing remain challenging problems for the compiler.

The invited speaker at CC 2001 was Ole Lehrman Madsen, whose talk was entitled *Virtual Classes and Their Implementation*. An abstract of the invited talk opens these proceedings.

The work of the CC 2001 program committee was conducted entirely by electronic means. We used the START conference management software from the University of Maryland. This proved to be very supportive for the work of the PC. Christian Probst did a remarkable job in setting it up, adding more functionality, and managing the technicalities of the submission and the reviewing process.

I am glad to acknowledge the hard work and friendly cooperation of the members of the program committee. I also wish to thank the much larger number of additional reviewers who helped us to read and evaluate the submitted papers. I appreciated very much the support and advice of the ETAPS chair, Don Sannella. Finally, I wish to thank all the authors of submitted papers for their continued interest, without which the CC conference could not thrive.

January 2001

Reinhard Wilhelm

Program Committee

Uwe Assmann (Karlsruhe)	Lex Augusteijn (Eindhoven)
David Bernstein (Haifa)	Stefano Crespi-Reghezzi (Milano)
Evelyn Duesterwald (Cambridge, USA)	Christine Eisenbeis (Rocquencourt)
Andreas Krall (Vienna)	Xavier Leroy (Rocquencourt)
Rainer Leupers (Dortmund)	Borivoj Melichar (Prague)
Mikael Petersson (Uppsala)	Tom Reps (Madison)
Martin Rinard (Cambridge, USA)	Reinhard Wilhelm (Saarbrücken)
	(Chair)

Referees

Giovanni Agosta	Paul F. Hoogendijk	Sara Porat
Pierre Amiranoff	Jan Hoogerbrugge	Christian Probst
Denis Barthou	Daniel Kaestner	Ganesan Ramalingam
Miroslav Benes	Felix Klock	Martin Rinard
Francois Bodin	Elliot K. Kolodner	Erven Rohou
Boris Boesler	Jaroslav Kral	Radu Rugina
Pierre Boullier	Viktor Kuncak	Petr Saloun
Alessandro Campi	Patrick Lam	Pierluigi San Pietro
Zbigniew Chamski	Marc Langenbach	Bernhard Scholz
Albert Cohen	Sam Larsen	Helmut Seidl
Giuseppe Desoli	Sylvain Lelait	André Seznec
Damien Doligez	Florian Liekweg	Dafna Shenwald
A. Ealan	Goetz Lindenmaier	Ron Sivan
Erik Eckstein	Vassily Litvinov	Mark Stephenson
Anton Ertl	Andreas Ludwig	Henrik Theiling
Paolo Faraboschi	Evelyne Lutton	Stephan Thesing
Paul Feautrier	Willem C. Mallon	François Thomasset
Stefan Freudenberger	Luc Maranget	Sid Ahmed Ali Touati
Hans van Gageldonk	Darko Marinov	Joachim A. Trescher
Thilo Gaul	Vincenzo Martena	Mon Ping Wang
Daniela Genius	Florian Martin	Rik van de Wiel
David Grove	Nicolay Mateev	Wim F.D. Yedema
Mustafa Hagog	Bilha Mendelson	
Dirk Heuzeroth	Andrea Ornstein	

Table of Contents

Invited Talk

Virtual Classes and Their Implementation.....	1
<i>Ole Lehrman Madsen</i>	

Program Analysis

Alias Analysis by Means of a Model Checker	3
<i>Vincenzo Martena, Pierluigi San Pietro</i>	
Points-to and Side-Effect Analyses for Programs Built with Precompiled Libraries	20
<i>Atanas Rountev, Barbara G. Ryder</i>	
A Novel Probabilistic Data Flow Framework	37
<i>Eduard Mehofer, Bernhard Scholz</i>	

Program Transformation

Imperative Program Transformation by Rewriting.....	52
<i>David Lacey, Oege de Moor</i>	
Compiler Transformation of Pointers to Explicit Array Accesses in DSP Applications.....	69
<i>Björn Franke, Michael O’Boyle</i>	
User-Extensible Simplification—Type-Based Optimizer Generators	86
<i>Sibylle Schupp, Douglas Gregor, David Musser, Shin-Ming Liu</i>	
A Practical, Robust Method for Generating Variable Range Tables	102
<i>Caroline Tice, Susan L. Graham</i>	

Program Analysis

Efficient Symbolic Analysis for Optimizing Compilers.....	118
<i>Robert A. van Engelen</i>	
Interprocedural Shape Analysis for Recursive Programs	133
<i>Noam Rinetzky, Mooly Sagiv</i>	
Design-Driven Compilation.....	150
<i>Radu Rugina, Martin Rinard</i>	

Intraprocessor Parallelism

Software Pipelining of Nested Loops	165
<i>Kalyan Muthukumar, Gautam Doshi</i>	
A First Step Towards Time Optimal Software Pipelining of Loops with Control Flows	182
<i>Han-Saem Yun, Jihong Kim, Soo-Mook Moon</i>	
Comparing Tail Duplication with Compensation Code in Single Path Global Instruction Scheduling	200
<i>David Gregg</i>	
Register Saturation in Superscalar and VLIW Codes	213
<i>Sid Ahmed Ali Touati</i>	

Parsing

Directly-Executable Earley Parsing	229
<i>John Aycock, Nigel Horspool</i>	
A Bounded Graph-Connect Construction for LR-regular Parsers	244
<i>Jacques Farré, José Fortes Gálvez</i>	

Memory Hierarchy

Array Unification: A Locality Optimization Technique	259
<i>Mahmut Taylan Kandemir</i>	
Optimal Live Range Merge for Address Register Allocation in Embedded Programs	274
<i>Guilherme Ottoni, Sandro Rigo, Guido Araujo, Subramanian Rajagopalan, Sharad Malik</i>	
Speculative Prefetching of <i>Induction Pointers</i>	289
<i>Artour Stouthchinin, José Nelson Amaral, Guang R. Gao, James C. Dehnert, Suneel Jain, Alban Dowillet</i>	
Constant-Time Root Scanning for Deterministic Garbage Collection	304
<i>Fridtjof Siebert</i>	

Profiling

Goal-Directed Value Profiling	319
<i>Scott Watterson, Saumya Debray</i>	
A Framework for Optimizing Java Using Attributes	334
<i>Patrice Pominville, Feng Qian, Raja Vallée-Rai, Laurie Hendren, Clark Verbrugge</i>	

Demos

SmartTools: A Generator of Interactive Environments Tools	355
<i>Isabelle Attali, Carine Courbis, Pascal Degenne, Alexandre Fau, Didier Parigot, Claude Pasquier</i>	
Visual Patterns in the VLEli System	361
<i>Matthias T. Jung, Uwe Kastens, Christian Schindler, Carsten Schmidt</i>	
The ASF+SDF Meta-environment: A Component-Based Language Development Environment	365
<i>M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, J. Visser</i>	
Author Index	371

Virtual Classes and Their Implementation

Ole Lehrmann Madsen

Computer Science Department, Aarhus University,
Åbogade 34, DK-8200 Århus N, Denmark

Ole.L.Madsen@daimi.au.dk

Abstract. One of the characteristics of BETA [4] is the unification of *abstraction* mechanisms such as class, procedure, process type, generic class, interface, etc. into one abstraction mechanism: the *pattern*. In addition to keeping the language small, the unification has given a systematic treatment of all abstraction mechanisms and leads to a number of new possibilities.

One of the interesting results of the unification is the notion of *virtual class* [7,8], which is the BETA mechanism for expressing genericity. A class may define an attribute in the form of a virtual class just as a class may define an attribute in the form of a virtual procedure. A subclass may then refine the definition of the virtual class attribute into a more specialized class. This is very much in the same way as a virtual procedure can be refined - resulting in a more specialized procedure. Virtual classes can be seen as an object-oriented version of generics. Other attempts to provide genericity for OO languages has been based on various forms of parametric polymorphism and function application rather than inheritance. Virtual classes have been used for more than 15 years in the BETA community and they have demonstrated their usefulness as a powerful abstraction mechanism. There has recently been an increasing interest in virtual classes and a number of proposals for adding virtual classes to other languages, extending virtual classes, and unifying virtual classes and parameterized classes have been made [1,2,3,13,14,15,16,17].

Another distinguishing feature of BETA is the notion of *nested class* [5]. The nested class construct originates already with Simula and is supported in a more general form in BETA. Nested classes have thus been available to the OO community for almost 4 decades, and the mechanism has found many uses in particular to structure large systems. Despite the usefulness, mainstream OO languages have not included general nesting mechanisms although C++ has a restricted form of nested classes, only working as a scoping mechanism. Recently nested classes has been added to the Java language.

From a semantic analysis point of view the combination of inheritance, and general nesting adds some complexity to the semantic analysis, since the search space for names becomes two-dimensional. With virtual classes, the analysis becomes even more complicated – for details see ref. [10].

The unification of class and procedure has also lead to an *inheritance mechanism for procedures* [5] where method-combination is based on the **inner**-construct known from Simula. In BETA, patterns are *first-class values*, which implies that procedures as well as classes are first-class values. BETA also supports the notion of *class-less* objects, which has been adapted in the form of anonymous classes in Java. Finally, it

might be mentioned that BETA supports *coroutines* as well as *concurrent active objects*. For further details about BETA, see [9, 11]. The Mjølner System is a program development environment for BETA and may be obtained from ref. [12].

References

1. Bruce, K., Odersky, M., Wadler, P.: A Statically Safe Alternative to Virtual Types, In: Jul, E. (ed.): 12th European Conference on Object-Oriented Programming, Brussels, June 1998. Lecture Notes in Computer Science, Vol. 1445. Springer-Verlag, Berlin Heidelberg New York, 1998.
2. Ernst, E.: Propagating Class and Method Combination, In: Guerraoui, R. (ed.): 13th European Conference on Object-Oriented Programming, Lisbon, June 1999. Lecture Notes in Computer Science, Vol. 1628. Springer-Verlag, Berlin Heidelberg New York, 1999.
3. Igarashi, A., Pierce, B.: Foundations for Virtual Types, In: Guerraoui, R. (ed.): 13th European Conference on Object-Oriented Programming, Lisbon, June 1999. Lecture Notes in Computer Science, Vol. 1628. Springer-Verlag, Berlin Heidelberg New York, 1999.
4. Kristensen, B.B., Madsen, O.L., Møller-Pedersen, B., Nygaard, K.: Abstraction Mechanisms in the BETA Programming Language. In Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages. Austin, Texas, January 1983.
5. Kristensen, B.B., Madsen, O.L., Møller-Pedersen, B., Nygaard, K.: Classification of Actions or Inheritance also for Methods. In: Bézivin, et al. (eds.): 1st European Conference on Object-Oriented Programming, Paris, June 1987. Lecture Notes in Computer Science, Vol. 276. Springer-Verlag, Berlin Heidelberg New York, 1987.
6. Madsen, O.L., Møller-Pedersen, B., Nygaard, K.: Object-Oriented Programming in the BETA Programming Language. Addison Wesley/ACM Press, Wokingham, England, 1993. (Out of print, but a reprint can be obtained from [12]).
7. Madsen, O.L., Møller-Pedersen, B.: Virtual Classes, a Powerful Mechanism in Object-Oriented Languages. In: Proc. OOPSLA'89, New Orleans, 1989.
8. Madsen, O.L., Magnusson, B., Møller-Pedersen, B.: Strong Typing of Object-Oriented Languages Revisited. In: Proc. OOPSLA'90, Ottawa, Canada, 1990.
9. Madsen, O.L.: Open Issues in Object-Oriented Programming — A Scandinavian Perspective. Software Practice and Experience, Vol. 25, No. S4, Dec. 1995.
10. Madsen, O.L.: Semantic Analysis of Virtual Classes and Nested Classes. In: Proc. OOPSLA'99, Denver, Colorado, 1999.
11. Madsen, O.L.: Towards a Unified Programming Language. In: Bertino, E. (ed.): 14th European Conference on Object-Oriented Programming. Lecture Notes in Computer Science, Vol. 1850. Springer-Verlag, Berlin Heidelberg New York, 2000.
12. The Mjølner System: <http://www.mjolner.com>
13. Shang, D.: Subtypes and Convertible Types, Object Currents, 1(6), June 1996.
14. Thorup, K.K.: Genericity in Java with Virtual Classes. In: Aksit, M., Matsouka, S. (eds.): 11th European Conference on Object-Oriented Programming. Lecture Notes in Computer Science, Vol. 1241. Springer Verlag, Berlin Heidelberg New York, 1997.
15. Torgersen, M.: Virtual Types are Statically Safe, 5th Workshop on Foundations of Object-Oriented Languages, January 1998.
16. Thorup, K.K., Torgersen, M.: Structured Virtual Types, informal session on types for Java, 5th Workshop on Foundations of Object-Oriented Languages, January 1998.
17. Thorup, K.K., Torgersen, M.: Unifying Genericity — Combining the benefits of Virtual types and parameterized types. In: Guerraoui, R. (ed.): 13th European Conference on Object-Oriented Programming, Lisbon, June 1999. Lecture Notes in Computer Science, Vol. 1628. Springer-Verlag, Berlin Heidelberg New York, 1999.

Alias Analysis by Means of a Model Checker⁺

Vincenzo Martena and Pierluigi San Pietro

Dipartimento di Elettronica e Informazione, Politecnico di Milano
P.za Leonardo da Vinci, 32. Milano 20133, Italia
{martena, sanpietr}@elet.polimi.it

Abstract. We study the application of a standard model checker tool, Spin, to the well-known problem of computing a may-alias relation for a C program. A precise may-alias relation can significantly improve code optimization, but in general it may be computationally too expensive. We show that, at least in the case of intraprocedural alias analysis, a model checking tool has a great potential for precision and efficiency. For instance, we can easily deal, with good precision, with features such as pointer arithmetic, arrays, structures and dynamic memory allocation. At the very least, the great flexibility allowed in defining the may-alias relation, should make it easier to experiment and to examine the connections among the accuracy of an alias analysis and the optimizations available in the various compilation phases.

1 Introduction

Two symbol or pointer expressions in a program are an *alias* when they reference the same memory location.

Alias analysis is the activity of detecting which expressions, at a given point in a program, are not aliases of each other. Static (i.e., compile-time) alias analysis is very important for generating efficient code [1], since many compiling optimizations rely on knowing which data could be referenced by a load or store expression [2,3]. Currently, in compilers for Instruction-Level Parallelism (ILP) processors alias analysis is even more important since it can improve the performance of the instruction scheduler [4]. However, exact alias analysis is impractical, and in general undecidable [5], because of the difficulties in determining which objects are referenced by pointers at a given point in a program. Hence, every approach to alias analysis makes some conservative approximations to the alias relation, determining what is called a *may-alias* relation. A pair of expressions $(e1, e2) \in \text{may-alias}$ in a given point of the program if a static analysis determines that "there is a chance" that $e1$ and $e2$ address the same memory cell at that point during some execution of the program. The relation must be *conservative*, i.e., if $(e1, e2) \notin \text{may-alias}$ then it is impossible that $e1$ and $e2$ may reference the same cell at that point: if this were not the case, a code optimization allocating different addresses to $e1$ and $e2$ would deliver an incorrect program. Clearly, the may-alias relation is an approximation that must be a compromise between preci-

⁺ Work partially supported by STMicroelectronics and by CNR-CESTIA.

sion (leading to efficient code) and the temporal and spatial efficiency of the computation.

Approximated static alias analysis has attracted a great body of literature and it is usually distinguished in flow-sensitive or flow-insensitive, context-sensitive or context-insensitive and interprocedural or intraprocedural [2]. In practice, even approximated solutions may be computationally expensive for large programs (e.g., see [6,7]): a precise analysis should be executed only after some other kind of imprecise-but-efficient analysis has shown the necessity of improving the precision for some parts of a program. It is also debatable how much precise an analysis should be to be considered really cost-effective in terms of optimization. The goal of our work is to build a tool that, rather than being adopted directly in a compiler, may be used to explore and assess the importance and usefulness of various approximations to the may-alias relation and also to check the results computed by other alias analysis tools.

The novelty of our approach is the adoption of a standard *model checking* tool, Spin [8]. A model checker is essentially a highly efficient analyzer of properties of finite state machines, using highly optimized concepts and algorithms, developed in years of study and experiments. In our approach, the relevant features of a C program are abstracted into a suitable finite-state automaton and the alias analysis problem is transformed into a reachability problem for the automaton. The advantage of this method is that no analysis algorithm has to be defined and implemented: the model checker takes care of the actual analysis, without having to program or develop any algorithm ourselves. Hence, our approach readily allows to study, extend, and experiment with various advanced features that are usually ignored in alias analysis [9,7,10]. For instance, the prototype we built so far is able to deal, with better precision than usual, with features such as dynamic memory allocation, aggregates (i.e., arrays and structs), multi-level pointers and pointer arithmetic in C.

Our approach to alias analysis may be especially useful in deciding whether a precise analysis effectively improves the performance of the code generated by a compiler for an ILP processor, by assessing whether the treatment of one or more precise features is really useful and worth incorporating in a compiler.

A long-term goal of our research is also the integration of our toolkit with other less precise analysis tools, such as the one being developed in our group [11] and others [12,13]. The model checking tool could then be used to check only those parts of a program that are found in need of greater optimization and thus require more precision in the analysis. Alternatively, if experiments with the model checker may show that a certain kind of precise analysis is feasible and useful for optimization, specific algorithms and tools may also be developed.

The paper is structured as follows. Section 2 briefly describes a model checker and introduces its usage in alias analysis on some short, but "difficult" examples. Section 3 summarizes the experimental results obtained so far, discussing issues of efficiency and extendibility. Section 4 draws a few conclusions and directions of future research.

2 Model Checking for Alias Analysis

Model Checking is the automated verification that a (often, finite state) machine, described in a suitable format, verifies a given property. The property must be described with a formal notation, which can be either the description of another machine or a temporal logic formula.

If the verification of the property fails, the Model Checker (MC) tool builds a counterexample, i.e., an execution trace of the specified system that leads to the violation of the desired property. By examining such traces, it is usually not difficult to identify the cause of the failure.

Model checking techniques have received great attention in the last years, due to the successes of the automatic verification of finite-state systems describing protocols, hardware devices, reactive systems. A symbolic MC [14] may routinely check systems with 10^{10} reachable states, and the case of 10^8 states is routine also for on-the-fly model checkers as Spin. In certain cases, much larger numbers have been obtained, but they usually correspond to less than a few hundreds bits of state space. This means that any non-trivial software system cannot be checked as it is. The approach for applying a MC to software programs and specifications is to use some form of *abstraction* to reduce the number of states. An abstraction omits many details, such as data structures, of the system to be checked. Finding the right abstraction is a very difficult problem, and there is in general no guarantee that the verification of the abstracted system brings any information about the real system. In case of alias analysis, however, we are mainly interested in pointer expressions and we can thus abstract away significant parts of the code to be checked, without loosing the correctness of the results. Also, various conservative approximations may be applied to the original code (e.g., replacing branch conditionals with nondeterministic choices), leading to small, and efficiently analyzable, finite state machines, but still providing a good precision of the analysis. The abstraction we propose is one of the main contributions of this paper.

2.1 The Spin Model Checker and the Promela Language

Spin is a widely distributed software package that supports the formal verification of distributed systems. For the sake of brevity, we do not include here a description of the tool itself and of its use, which is widely available also via web.

The input language of Spin is called Promela. We cannot describe here a language rich and complex such as Promela, whose description is widely available. The syntax of Promela is C-like, and we ignore here Promela's communication aspects. Conditional expressions may be defined as: `(expr1 -> expr2 : expr3)`, which has the value of `expr3` if `expr1` evaluates to zero, and the value of `expr2` otherwise. A special `skip` statement denotes the null operation, i.e., it does nothing (its use derives from the syntactic constraints of Promela that do not allow "empty" statements). Processes may be declared with a `proctype` declaration. We are only interested in declaring one process, `active proctype main()`, which basically corresponds to the `main()` function of a C program. Functions and procedures may be declared with an

inlining mechanism. The control structures are a selection (`if`) statement and a repetition (`do`) statement, but it is also possible to use labels and `goto` statements. Selection has the form: `if :: statements ... :: statements fi`. It selects one among its options (each of them starts with `::`) and executes it. An option can be selected if its first statement (the guard) is enabled (i.e., it evaluates to a number greater than 0). A selection blocks until there is at least one selectable branch. If more than one option is selectable, one will be selected at random. The special guard `else->` can be used (once) in selection and repetition statements and is enabled precisely if all other guards are blocked. Repetition has the form: `do :: statements ... :: statements od`. It is similar to a selection, except that the statement is executed repeatedly, until the control is explicitly transferred to outside the statement by a `goto` or a `break`. A `break` will terminate the innermost repetition statement in which it is executed.

2.2 Applying Model Checking to Alias Analysis on Demand

The application of the Spin model checker to alias analysis requires to deal with two distinct problems:

- 1) how to encode the program to be analyzed into the Promela language of Spin;
- 2) how to encode and retrieve the alias information.

Encoding C into Promela.

The problem of encoding C into Promela is essentially the issue of finding the right abstraction to solve the may-alias problem, since the model checker cannot deal exactly with all the data and variables used in a real C program. In alias analysis, the important task is to find aliasing information for pointer expressions: a good abstraction could remove everything not directly related to pointers (such as the actual data values). Actual address values (integers) may be dealt with by Spin, provided their value range is finite (32-bit integers are supported). Hence, each nonpointer variable in a program may be replaced by an integer constant, denoting its address. Also a pointer variable must have a static address, but it must also *store* an address: it corresponds to an integer variable (actually, an array of variables is introduced, one cell of the array for each pointer, to make it easier to reference/dereference dynamic structures and multi-level pointers).

Also, as it is usually the case in alias analysis, we can ignore the actual conditions in conditional branching instructions, by replacing them with nondeterministic choices, at least when they do not involve pointer expressions. Notice that Spin, when confronted with a nondeterministic choice among two alternatives, always keeps track that only one is actually taken. This is not often implemented in alias analysis. For instance, the C statement:

```
if (cond) {p1=&a; p2=&b;}
else {p1 = &b; p2 = &a;}
```

is translated into Promela with a nondeterministic choice between the two branches, but no mixing of the information is done: Spin does not consider `p1` and `p2` to be aliases.

We may also make Spin to deal with `malloc()` instructions, which generate a new address value and allocate new memory, by simulating this generation. However, since in general the number of executions of a `malloc` may be unbounded, we make the conservative assumption that each occurrence of a `malloc` in a C program may generate explicitly only a certain fixed number of actual addresses (e.g., just one), and, after that, the `malloc` generates a fictitious address that is a potential alias of every other address generated by the same occurrence of the `malloc` statement. Records (`struct`) types may be dealt as well, by allocating memory and computing offsets for the pointer fields of a new `struct` and by generating new addresses for the nonpointer fields.

Encoding and Retrieving the Alias Information.

The typical application of a model checker is to build counterexamples when a given property is violated, or to report that no violation occurs. However, for recovering the may-alias relation after verification with Spin, we need to have aliasing results rather than counterexamples. Our idea is to use a "byproduct" of model checking analysis, namely unreachability analysis. A model checker like Spin is able to report the unreachable states of a Promela program. Hence, we can add a simple test stating that a pair of pointer expressions are aliases of each other, followed by a null (`skip`) operation to be executed in case the test is verified: if the analysis shows that the null operation is unreachable in that point, then the two expressions cannot be aliases.

In this way, it is possible to compute the complete may-alias relation in each point of the program, even though this would mean adding a great number of pairs (test, null operation) to the Promela program, since the total number of states only increases linearly with the number of may-alias tests. It is however more natural and convenient to compute only the part of the may-alias relation that is deemed useful by a compiler.

2.3 The Translation Scheme

A simple translator is being designed to compile C programs into Promela. In this section, we illustrate the translation scheme by translating a few examples of simple, but not trivial, C programs. All the programs will be monolithic, i.e., not decomposed in functions. This is not a limitation, since nonrecursive function calls may be replaced by code inlining; however, the inlining could cause a significant slowdown for the interprocedural analysis of large programs.

Pointer Arithmetics and Static Arrays.

The first program, called `insSort.c`, shown in Fig. 1, is a simple insertion sort of an array of integers, using pointer arithmetics. The program reads 50 integers, store them in an array `numbers` and then prints them in increasing order. The line numbers are displayed for ease of reference.

For instance, we may be interested in computing alias information about `position` and `index` at the beginning and at the end of the inner loop (lines 13 and 18). The `int main()` declaration corresponds in Promela to the declaration `active proctype main()`. We now show how to deal with variable declarations. Each non-pointer variable declaration is replaced by the assignment of a progressive integer

constant (the address) to the variable name. The addresses conventionally start at one. For instance, `int key;` is replaced by: `#define key 1.`

Each pointer declaration corresponds to the Promela declaration of an integer variable. To allow a homogenous treatment of multi-level pointers and of dynamic memory allocation, an array of integers is declared, called `Pointers`: each pointer is an element of the array. Each pointer name is defined in Promela by an integer constant (the static address of the pointer), which is used to index the array. Since no dynamic memory allocation is used in this program, the dimension of this array is simply the number of pointers in the program, i.e., two. We use three instead, since for various reasons we prefer to ignore the first element of the array. Each pointer variable is then replaced by the constant index of the array.

Hence, the declarations: `int *index;` `int *position;` are replaced by the Promela declarations:

```

1.int main(){
2.  int key;
3.  int numbers[50];
4.  int *index;
5.  int *position;
6.  for (index=numbers; index<numbers+50; index++) /*read
   array */
7.    scanf("%d", index);
8.  index=numbers+1;
9.  while(index<numbers+50){ /*sort the array */
10.   key=*index;
11.   position=index;
12.   while (position>numbers){
13.    /*position and index are aliases here? */
14.    if ( *(position-1) > key) {
15.     *position= *(position-1);
16.    }
17.    position--;
18.    /*position and index are aliases here? */
19.   }
20.   *position=key;
21.   index++;
22.  }
23.  printf("Sorted array\n");
24.  for (index=numbers; index<numbers+50; index++) /*print
   array */
25.   printf("%d\n",*index);
26.}

```

Fig. 1. The `insSort.c` program: a simple insertion sort in a static array

```

int Pointers[3];
#define index 1;
#define position 2;

```

The actual address referenced by the pointer `index` is denoted with `Pointers[index]`, while the address `&index` is denoted by the name `index` itself. To make pointer expressions easier to write and generalize, we prefer to introduce a C-preprocessor macro, called `contAddr` ("content at the given address"), which returns the address stored in a pointer:

```
#define contAddr(expression) Pointers[expression]
```

Hence, `contrAddr(position)` corresponds in Promela to the address that in C is denoted by `position`, while `position` actually denotes in Promela what in C is denoted by `&position`.

An array declaration, such as `int numbers[maxEl]`, is considered as a declaration of a group of non-pointer variables, in this case `maxEl` variables. Hence, we allocate `maxEl` consecutive integer constants for the addresses of the array elements, without allocating any memory for the array itself. The array declaration is replaced by `#define numbers 2;`

Lines 6 and 7 only read values in the array and are thus ignored. The assignment of line 8 means that the address of the second cell of the array `numbers` (i.e., the address `numbers + 1`) is assigned as the new value of the pointer `index`. To make the Promela program more readable and extendable, we introduce a macro of the C pre-processor for this kind of assignments, called `setAddr`:

```
#define setAddr(P,expression) Pointers[P] = expression
```

Hence, `index=numbers + 1` becomes `setAddr(index,numbers+1)`. The external while loop of the insertion sort (line 9) must be replaced by a Promela loop. Hence, a `while(C) B;` statement is replaced by the Promela statement: `do ::P(C)-> P(B); ::else -> break; od` where `P(C)` and `P(B)` are the Promela translation of the conditional `C` and of the (possibly compound) statement `B`, respectively. In this case, the condition is translated to `(contrAddr(position)>numbers)`

The assignment statement of line 10 is ignored, since no pointer value is involved, while the one of line 11 (between pointers) becomes `set(position,index);` where `set` is defined by the following C-preprocessor macro:

```
#define set(P,ex) setAddr(P,contAddr(ex))
```

Both `set` and `setAddr` assign a value to the cell indexed by the first argument `P`, but `setAddr` considers this value to be directly the second argument, while `set` considers this value to be the content of the cell whose address is the value of the second argument. Hence, the C fragment: `int a; int *p, *q; p = &a; q=p;` is translated into: `setAddr(p,a); set(q,p);`

The conditional `if` statement of lines 14 to 16 can be eliminated. In fact, the condition cannot be computed on pointer values only and should be replaced by a non-deterministic Promela `if` statement. However, since line 15 must be eliminated as well, the translation of the conditional would be:

```
if ::true -> skip; ::true -> skip; fi
```

which obviously has no effect. Line 17 becomes `Pointers[index]++;` or, alternatively, `setAddr(index,contAddr(index)+1)`, and the other lines may be ignored.

The alias information is introduced, in the points corresponding to the lines 13 and 18, by using a simple test, defined via a pair of macro as follows:

```
#define mayAlias(X,Y) if ::(Pointers[X] == Pointers[Y]) ->
skip;
#define notAlias      ::else-> skip; fi
```

The complete translation of the C program above is reported in Fig. 2.

When Spin is run to analyze the program, it reports that the `mayAlias` condition of line 16 is the only unreachable part of the code. Hence, `position` and `index` do not belong to the may-alias relation at line 18 of the original program. Instead, since the other `mayAlias` condition is reachable, the two pointers do belong to the may-alias relation at line 13 of the original program. Notice that in the latter case the relation is not a must-alias relation (since also the `notAlias` condition is reachable as well). The relationship between the speed of Spin in providing the answer, the total memory usage and the size of the static array is reported in Section 3.

Multi-level Pointers.

Multi-level pointers can be easily dealt with. For instance, a pointer declared as `int** p`; is declared in Spin again as a static address for `p`, used as an index of the pointer array. The operator `contAddr` can be used to dereference a pointer expression. Consider the following fragment of code:

```
char **p2; char * p1; char a;
...
*p2 = &a; p2 = &p1; *p2 = p1; p1 = *p2;
...
```

This can be easily translated into Promela:

```
#define a 1
#define p1 2
#define p2 3
...
setAddr(contAddr(p2), a);
setAddr(p2,p1);
set(contAddr(p2),p1);
set(p1, contAddr(p2));
```

Structures and Dynamic Memory Allocation.

Traditionally, in alias analysis, programs with dynamic memory allocation are either ignored or dealt with in a very limited way: when a `p = (T*) malloc(sizeof(T));` instruction occurs in a program, the expression `*p` is considered to be an alias of every other pointer expression of type `T`. At most, some separate treatment is introduced by distinguishing among the pointers initialized with `malloc` instructions at different lines of the program [7,9]. This is a conservative assumption, assuring that the may-aliases relation always includes every pair of pointer expressions that are aliases, but it is often too imprecise. For instance, if `T` is `struct {int`

info; T* next;}, the pointer expression $p \rightarrow \text{next}$ is assumed to be an alias of every pointer expression of type T^* .

```

1. #define maxEl 50
2. #define max_element 3 /*the number of pointers
   declared in the program*/
3. #define key 1
4. #define numbers 2
5. #define index 1
6. #define position 2
7. int Pointers[max_element];
8. active proctype main () {
9.     setAddr(index,numbers+1);
10.    do      ::contAddr(index) < (numbers+maxEl) ->
11.            set(position,index);
12.        do ::(contAddress(position)>numbers) ->
13.            may_alias(position,index)
14.            notAlias;
15.            contAddr(position)--;
16.            mayAlias(position,index)
17.            notAlias;
18.        ::else -> break;
19.    od;
20.    contAddr(index)++;
21.    ::else -> break;
22. od
23.}

```

Fig. 2. The Promela translation of insSort.c

For a more complete example, consider the simple program `doubleList.c` of Fig. 3, which inserts two integers in a double-linked list built from scratch.

```

1. struct elem {int inf; struct elem next*; struct elem previ-
   ous*};
2. int main(){
3.     struct elem *root;
4.     root = (struct elem*)malloc(sizeof(struct elem));
5.     scanf("%d", &(root->inf));
6.     root->previous = null;
7.     root->next = (struct elem*)malloc(sizeof(struct elem));
8.     root->next->previous=root;
9.     root->next->next = null;
10.    scanf("%d", &(root->next->inf));
11.    /* root->next and root->next->previous may be alias here?*/
12.}

```

Fig. 3. The `doubleList.c` program, which inserts two elements in a double-linked list

In a traditional approach, $\text{root} \rightarrow \text{next}$ and $\text{root} \rightarrow \text{next} \rightarrow \text{previous}$ are all considered to be aliases, preventing optimizations. We will show that this is not the case in our approach.

To make this presentation simple, here we show how to deal with `malloc` instructions combined with variables of type `struct*`, ignoring statically-declared variables of type `struct` and dynamic allocation of pointers to values of the basic types. Also, we assume that the address of a nonpointer field of a structure is never taken, i.e., there is no statement of type `x = &(root->inf)`. The latter restriction allows us to ignore the non-pointer fields, since in this case no pointer expression may be an alias of a nonpointer field. Notice that the actual translation we implemented does not have any of these limitations, which can be relaxed without any loss in efficiency by using a more complex translation. Another limitation is that we do not allow casting of structure types (as instead done in [15]), since we treat each pointer field as a separate object based on its offset and size: the results would not be portable because the memory layout of structures is implementation-dependent.

The declaration of the `struct` of line 1 corresponds to the following Promela declaration:

```
#define sizeof_elem 2 /*number of pointer fields in elem */
#define offset_elem_next 0 /*offset of "next" field */
#define offset_elem_previous 1 /*offset of "previous" field
*/
```

Number all the `malloc` statements in the source code, starting from 1. We introduce in Promela a `malloc` statement called `malloc(P,T,N)`, where `P` is a pointer expression of type `T*`, `T` is the type name and `N` is the `malloc` number. Hence, a statement of the form `p = (int*) malloc(sizeof(int));` (corresponding to the 3rd occurrence of a `malloc` in the source code) is translated in Promela into `malloc(p,int,3);`

A `malloc(P,T,N)` statement must generate a new address for a variable of type `T` and store it in the cell of the `Pointers` array whose index is `P`. To be able to define dynamically new addresses, the size of the array `Pointers`, defined via the constant `max_element`, must be greater than the number of pointer variables declared in the program. A new address can be easily generated by declaring a global counter variable, called `current`, to be incremented of `sizeof_T` each time a new address is allocated (and to be checked against `max_element`). The new address may again be used to index the array `Pointers`. Also, there is a global array `count_malloc` of integer counters, which keeps count of the number of actual allocations for each `malloc`. Only a limited number of addresses, denoted by the constant `max_malloc`, is in fact available for each `malloc` (typically, just one): hence, if no new address is available (`count_malloc[N] >= max_malloc`), the `malloc` must return a fictitious address, whose value is conventionally `all_alias + N`, where `all_alias` is a constant denoting a special, fictitious address that is used to signal that an expression address is actually an alias of every other expression. Every address greater or equal to `all_alias` is considered to be fictitious, i.e., it does not refer to an actual memory cell in the Promela program (hence, `all_alias` must be greater than `max_element`). The `malloc` statement in Promela is defined as follows (with the C preprocessor, `"\n"` is used to define a multi-line macro, while `###` denotes string concatenation)

```
int current=total_number_of_pointer_variables+1;
```



```

int Pointers[max_element];
byte count_malloc[total_number_of_malloc];
#define malloc(P,T,n) \
    if \
    :: P >= all_alias -> skip; \
    :: else -> \
        if \
        :: (current + sizeof_##T <= max_element)&& \
           count_malloc[n-1] < max_malloc -> \
           Pointers[P] = current; \
           current=current+sizeof_##T; \
           count_malloc[n-1]++; \
        :: else -> Pointers[P]=all_alias+n; \
    fi; \
fi

```

P stands for the address (index) of the memory cell where the newly generated address must be stored. Hence, the test `P >= all_alias -> skip;` is introduced in order not to allocate memory to a nonexistent cell. If the test is false, the `malloc` has to check whether there is space enough in the unused portion of the array `Pointers` to allocate `sizeof_T` consecutive cells (`current + sizeof_T <= max_element`) and whether the maximum numbers of addresses available for that single `malloc` has not been exceeded (`count_malloc[n-1] < max_malloc`). If the test is passed, the cell of index `P` is assigned the new address, the current counter is incremented of `sizeof_T`, and `count_malloc[n-1]` is incremented of one. Otherwise, the fictitious address `all_alias+n` is stored in `Pointers[P]`, to denote that the `n`-th `malloc` has not been able to allocate a new address.

To be able to deal with structures and their fields, we need to introduce Promela equivalents of `root->previous`, `&(root->next)`, etc. We first modify the operator `contAddr` to be able to deal with fictitious addresses:

```

#define contAddr(expr) (expr >= all_alias -> expr : Pointers[expr])

```

Hence, if the address of `expr` is fictitious, the array `Pointers` is not accessed and the fictitious value is returned.

The address of a field of a pointer expression `expr` of type `T` is given by the Promela macro `getField`:

```

#define getField(expr,field,T) (contAddr(expr)>=all_alias ->
all_alias :\
    Pointers[expr]+offset_##T##_## field) \

```

Hence, the access of a field returns the `all_alias` value if its address is fictitious, the address of the field otherwise. The latter address is obtained by adding, to the address of the first cell of the structure, the value of the offset of the field inside the structure. Hence, line 7 of the above C program:

```

root->next = (struct elem*)malloc(sizeof(struct elem));

```

is denoted in Promela by:

```

malloc(getField(root,next,T), T,2);

```

When a field is assigned a new value, e.g., `root->previous = null`; it is not possible to use the `setAddr` operator defined above, since now the various pointer expressions involved may also assume fictitious values. Hence, we redefine the operator as follows:

```
#define setAddr(address,expr) if \
    ::address >= all_alias -> skip; \
    ::else -> Pointers[address]=expr; \
fi
```

When a pointer expression of address `address` is assigned the value `expr`, we first check whether `address` denotes a valid cell, otherwise no assignment can be executed.

Hence, `root->previous = null`; is translated into:

```
setAddr(getField(root, previous), null);
```

where the null pointer is simply the constant 0: `#define null 0`

The `mayAlias-notAlias` pair has now to be extended: in fact, two pointer expressions may be alias also because either one evaluates to `all_alias`.

We extend the operators as follows:

```
#define mayAlias(ex1, ex2) if \
    :: (contAddr(ex1) == contAddr(ex2) && \
        contAddr(ex1) != null) -> skip; \
    :: (contAddr(ex1) == all_alias || \
        contAddr(ex2) == all_alias) -> skip
#define notAlias :: else -> skip; fi
```

The `mayAlias` operator checks whether the two expressions `ex1` and `ex2` reference the same cell, provided that the cell is valid and that the expressions do not refer to the null pointer. Notice that two cells with invalid but different addresses, such as `all_alias+1` and `all_alias+2`, are not considered to be aliases of each other (since they were generated by two distinct `malloc` instructions). Otherwise, the `mayAlias` operator checks whether one of the two expressions is an `all_alias`: in this case, the two expressions are also considered potential aliases. The `notAlias` operator corresponds to the case where neither of the two previous cases occurs: the two expressions cannot be aliases. The complete translation of the `doubleList.c` program is reported at <http://xtese1.elet.polimi.it>.

The result of the analysis with Spin is that the only unreachable part is the `mayAlias` condition of line 16, meaning that, as expected, `root->next` and `root->next->previous` cannot be aliases. Since the program is quite small, the execution time and memory requirements of Spin are negligible.

3 Experimental Results

Table 1 summarizes the experimental results obtained for the example `insSort.c` of Section 2. The running times are given on a PC with a Pentium III 733 MHz processor, with 256 KB cache Ram, 256 MB Ram and the Linux OS (vkernel 2.4.0, glibc

Pointers Elements	Array El.	State Vector (byte)	States	Memory Usage (Mbyte)	Approximated Analysis	Search Depth	Running Time (sec)
4	40	28	5085	2.527	No	5084	0.01
4	80	28	19765	3.192	No	19764	0.06
4	160	28	77925	5.861	No	77924	0.24
4	320	28	309445	16.501	No	309444	1.07
4	640	28	1.23e+06	62.716	No	1.233e+06	4.68
4	1000	28	3.00e+06	151.140	No	3.000e+06	35.6

Table 1. Performance results for `InsSort.c`

2.1). Notice that the running times do not include the compilation time (Spin generates a C program to be compiled with gcc before execution), which usually is fairly independent on the size of the Promela program and takes a few seconds. Memory occupation is often dominated by the stack size, which must be fixed before the execution. Hence, if a stack size too large is chosen, the memory occupation may seem quite large even if only a few states were reachable. On the other hand, the exploration of a large number of states may be very memory consuming, even though often can be completed in a very short time.

As it can be noticed, the number of states explored by Spin with `insSort.c` is quadratic in the size of the static array, but the running time is still small even when the array has hundreds of elements. The quadratic complexity is not surprising, since it results from the time complexity of the original insertion sort algorithm. The memory occupation is large, and it is dominated by the size of the stack used by Spin, while the number of bits in each state (called *state vector size* in Spin) is negligible, since the array size only impacts on the number of reachable states.

We also explored the use of Spin using "critical" examples, to check whether this approach really improves precision of analysis on known benchmarks. We considered, among others, one example taken from [7]. The example was introduced by Landi in order to find a "difficult" case where his new algorithm for alias analysis with dynamic memory allocation performed poorly. Landi's original source code and its Promela translation are reported at <http://xtese1.elet.polimi.it>. This example represents the worst case for Landi's algorithm, which finds $3n^3 + 7n^2 + 6n + 18$ aliases, where the parameter n is the size of the array v . Our analysis, instead, finds the exact solution, $n+11$, for each fixed n . Notice that our analysis is also able to distinguish that b and d cannot be alias after the execution of the two conditionals inside the `while` loop. Table 2 shows also the performance of the tool for various values of the parameter n . The running time is still quadratic, even though the original program runs in linear time. In fact, both the number of states searched and the state vector size increase linearly: the memory occupation and the running time must be a quadratic function of n .

Array Elements	State Vector (byte)	States	Memory Usage (Mbyte)	Approximated Analysis	Search Depth	Running Time (sec) (compilation excluded)
40	432	754	2.302	No	387	0.01
80	432	1394	2.404	No	707	0.03
160	832	2674	2.849	No	1347	0.09
320	1632	5234	3.822	No	2627	0.36
1000	4112	16114	11.557	No	8067	2.92
2000	8112	32114	36.957	No	16067	11.43
4000	16112	64114	135.757	No	32067	46.00

Table 2. Performance Results for Landi's example

Another example of a nontrivial program has been translated into Promela and the quality of the alias analysis has been assessed and the performance results have been studied. The example takes in input two series of integers in two separate linked lists, calculates the maximum element of the two lists and then stores their sum in a third list. The source code `LinkedLists.c`, along with its translation, is reported at <http://xtese1.elet.polimi.it>, where we will collect further experimental results. The example is composed of 116 lines of C code and makes a heavy use of dynamic memory allocation to insert elements in the various lists. The tool is able to distinguish as not being aliases at least the heads of the tree lists, even though the analysis cannot be exact due to the presence of malloc instructions inside unbounded loops. Since each malloc instruction is allowed to generate only a very limited number of new non-fictitious addresses, a very small number of pointers can be allocated dynamically by the Promela version. Hence, the maximum number of dynamically allocated pointers does not affect the performances.

This and the above results show that the running time and memory occupation do seem to depend just on the sheer size of the program to be analyzed, but especially on the number of variables. Notice that the memory occupation could be considerably reduced in most examples, by reducing the size of the available addresses (now, 32-bit integers).

4 Conclusions and Related Works

In this paper, we presented a prototype tool, based on the model checker Spin, to study and experiment with alias analysis. There are many known algorithms and tools that can be used to determine may-alias relations, with varying degrees of accuracy (for papers including broad references to alias analysis see for instance [16]). At the best of our knowledge, we ignore of any previous application of model checker technology to this goal. However, there are various studies about the application of model checking to flow analysis, e.g. [17]. In these works, it has been shown how many problems usually solved by means of data-flow techniques can be solved more simply by model checking techniques. In particular, our work is consistent with the methodology proposed in [18], which uses abstract interpretation [19] to abstract and analyze programs by combining model checking and data flow techniques. The links among abstract interpretation and data-flow analysis are also well-known (e.g., [20]). There

is also some relation with works, such as [21], that transform program analysis (e.g., flow-insensitive point-to analysis), into graph reachability problems (on context-free languages), since in our approach the may-alias relation is computed by using the related concept of state reachability (on finite-state automata). Our approach has also connections with works rephrasing static analysis questions in terms of systems of constraints (such as the application of the Omega-library in [22]). There are different approaches that try to deal with dynamic memory allocation. For instance, [23] performs very precise or even exact interprocedural alias analysis for certain programs that manipulate lists, using a symbolic representation. However, the result only has theoretical relevance, since it fails to scale up to real cases. Another method is shape analysis [24], that gives very good results in dealing with acyclic dynamic structures, but does not appear to deal with the cyclic case, such as the double linked lists studied in this paper.

By using our tool, we were able to experiment with more precise may-alias relations than are usually considered in intraprocedural analysis, allowing the study of programs including features such as pointer arithmetics, structures, dynamic allocation, multi-level pointers and arrays. Preliminary experiments with the tool show that it could have better performances even than specialized tools for precise analysis, but further experiments are required to assess this claim. Future work will apply the tool to standard benchmarks and C program, after having completed a translation tool from C to Promela. We are going to apply the tool to object oriented languages such as Java and C++ and to study other applications to static analysis of programs (such as detecting bad pointer initialization). An advantage of our approach is that all the experiments can be easily performed without writing any algorithm, but only defining a few operators and leaving all the processing job to the highly optimized Spin model checker.

The computation of the may-alias relation with a model checker is consistent with an approach to alias analysis and program optimization called *alias on demand* [11]. The approach is based on the consideration that in the optimization phase not all elements in the may-alias relation have the same importance: first, only those parts of the code that are executed frequently (such as loops) may deserve optimization, and hence the may-alias relation for the other parts is not useful; second, many parts of the may-alias relation do not enable any optimization. Therefore, in this approach the may-alias relation is not computed entirely before the optimizations are performed, but only after a preliminary optimization analysis has determined that a certain set of instructions could be optimized if there is no aliasing among certain pointer expressions. For instance, an optimization such as code parallelization for ILP processors must rely on limited parallel resources: most of the may-alias relation is completely irrelevant, since we cannot parallelize many instructions anyway. Therefore, it is possible to save a great deal of computational power that is otherwise needed for alias analysis.

For the moment, interprocedural analysis with our tool must be done by inlining the procedure calls in the main. Hence, precise analyses of the kind described here are probably infeasible for large programs. However, our approach could be easily tailored to deal with the precision that is required for the problem at hand. For instance, loops could be transformed in conditional statements and further details of a program could be omitted, leading to a less precise, but more efficient, analysis when and

where precision is not essential. Again, this could be obtained with almost no cost, leaving the user of the tool the possibility of customizing the analysis to the level of precision that is required. An alternative is to use infinite-state model checking, which however is still a subject of intense research.

A major goal of our group at Politecnico di Milano is to analyze the benefits that can be obtained by using alias information in the various compilation phases (register allocation, instruction scheduling, parallelization) and to examine the connections among the accuracy of the required alias analysis and the intended optimizations [1], also in view of the machine architecture. We believe that our tool could easily be tailored to support this kind of analysis.

Acknowledgements. We thank Marco Garatti and Giampaolo Agosta for their help and their comments. Special thanks to Stefano Crespi Reghizzi for making this research possible by providing us with ideas, encouragement and suggestions.

References

1. M. Shapiro and S. Horwitz, *The effects of the precision of pointer analysis*, In P. Van Hentenryck, (ed.), 4th Intern. Static Analysis Symp., 1997, LNCS 1302, pp. 16-34. Springer-Verlag.
2. S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann, 1997.
3. D. Bacon S. Graham and O. Sharp, *Compiler Transformations for High-Performance Computing*, ACM Computing Surveys, Vol. 26, n. 4, 1994, 345-419.
4. Joseph A. Fisher, *Trace Scheduling: A Technique for Global Microcode Compaction*, IEEE Transactions on Computers, 1981, Vol. 30, n. 7, 478—490.
5. G. Ramalingam, *The Undecibility of Aliasing*, ACM TOPLAS, Vol. 16, n. 5, 1994, 1467-1471.
6. S. Horwitz, *Precise Flow-Insensitive May-Alias Analysis is NP-Hard*, ACM TOPLAS, Vol. 19, n. 1, 1997, 1-6.
7. W. A. Landi, *Interprocedural Aliasing in the Presence of Pointers*, Rutgers Univ., PhD Thesis, 1997.
8. G. Holzmann, *Design and Validation of Computer Protocols*, Prentice-Hall, Englewood Cliffs, New Jersey, 1991.
9. W. Landi and B. Ryder, *A Safe Approximate Algorithm for Interprocedural Pointer Aliasing*, Prog. Lang. Design and Impl. ACM SIGPLAN Not., 1992, 235-248.
10. D. Liang and M.J. Harrold, *Equivalence Analysis: A General Technique to Improve the Efficiency of Data-flow Analyses in the Presence of Pointers*, ACM TOPLAS, Vol. 24, n. 5, 1999, 39-46.
11. M. Garatti, S. Crespi Reghizzi, *Backward Alias Analysis*, Tech.Report, Dip. di Elettronica E Informazione, Politecnico di Milano, Sept. 2000.
12. B. Steensgaard, *Points-to Analysis in Almost Linear Time*, in Proc. 23rd SIGPLAN-SIGACT Symp. on Principles of Programming Languages, Jan., 1996.pp. 32--41, ACM Press.
13. D. Liang and M.J. Harrold, *Efficient Points-to Analysis for Whole-Program Analysis*, Lectures Notes in Computer Science, 1687, 1999.
14. K. L. McMillan, *Symbolic model checking - an approach to the state explosion problem*, PhD thesis, Carnegie Mellon University, 1992.
15. Yong, S.H., Horwitz, S., and Repts, T., *Pointer analysis for programs with structures and casting*, Proc. of the ACM Conf. on Programming Language Design and Implementation, (Atlanta, GA, May 1-4, 1999), in ACM SIGPLAN Notices 34, 5 (May 1999), pp. 91-103.

16. M. Hind, M. Burke, P. Carini and J. Choi, *Interprocedural Pointer Alias Analysis*, ACM TOPLAS, Vol. 21, 4, 1999, 848-894.
17. B. Steffen, *Data Flow Analysis as Model Checking*, Proc. Int. Conf. on Theoretical Aspects of Computer Software, (TACS 1991), Sendai (Japan), September 1991, pp. 346-365.
18. D.A. Schmidt and B. Steffen, *Program analysis as model checking of abstract interpretations*, G. Levi. (ed.), pages 351--380. Proc. 5th Static Analysis Symp., Pisa, September, 1998. Berlin: Springer-Verlag, 1998. Springer LNCS 1503.
19. Cousot, P. and Cousot, R., *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*, POPL 1977, pp. 238--252.
20. Cousot, P. and Cousot, R., *Temporal Abstract Interpretation*, POPL 2000, pp.12-25.
21. Reps, T., *Program analysis via graph reachability*, Information and Software Technology 40, 11-12 (Nov./Dec. 1998), pp. 701-726.
22. W. Pugh and D. Wonnacott, *Constraint-Based Array Dependence Analysis*, ACM TOPLAS, Vol. 20, n. 3, 1998, 635-678.
23. Alain Deutsch, *Interprocedural May-Alias Analysis for Pointers: Beyond k-limiting*. In PLDI 1994, pp. 230-241. Jun 1994.
24. Wilhelm, R., Sagiv, M., and Reps, T., *Shape analysis*. In Proc. of CC 2000: 9th Int. Conf. on Compiler Construction, (Berlin, Ger., Mar. 27 - Apr. 2, 2000).

Points-to and Side-Effect Analyses for Programs Built with Precompiled Libraries

Atanas Rountev and Barbara G. Ryder

Department of Computer Science, Rutgers University, Piscataway, NJ 08854, USA
{rountev,ryder}@cs.rutgers.edu

Abstract. Large programs are typically built from separate modules. Traditional *whole-program analysis* cannot be used in the context of such modular development. In this paper we consider analysis for programs that combine client modules with precompiled library modules. We define *separate analyses* that allow library modules and client modules to be analyzed separately from each other. Our target analyses are Andersen's points-to analysis for C [1] and a side-effect analysis based on it. We perform separate points-to and side-effect analyses of a library module by using *worst-case assumptions* about the rest of the program. We also show how to construct *summary information* about a library module and how to use it for separate analysis of client modules. Our empirical results show that the separate points-to analyses are practical even for large modules, and that the cost of constructing and storing library summaries is low. This work is a step toward incorporating practical points-to and side-effect analyses in realistic compilers and software productivity tools.

1 Introduction

Large programs are typically built from separate modules. Such modular development allows better software management and provides a practical compilation model: instead of (re)compiling large programs from scratch, compilers can perform separate compilation of individual modules. This approach allows sharing of modules between programs; for example, an already compiled library module can be reused with no implementation or compilation cost. This development model also allows different modules to be developed by different teams, at different times and in separate locations.

Optimizing compilers and software productivity tools use static analyses to determine various properties of program behavior. Many of these analyses are performed by analyzing the whole program. However, such *whole-program analyses* cannot be used in the context of a modular development process. To make these analyses useful in real-world compilers and software tools, analysis techniques must be adapted to handle modular development.

This paper investigates one instance of this problem. We consider analysis for programs built with reusable *precompiled library modules*. Reusable modules are designed to be combined with many (yet unknown) clients, and are typically packaged as precompiled libraries that are subsequently linked with the client

code. We consider programs with two modules: a *library module* that is developed and compiled independently of any particular client, and a *client module* that uses the functionality of the library module.¹ For such programs, compilers cannot use whole-program analyses because the two modules are compiled separately. We show how certain whole-program points-to and side-effect analyses can be extended to handle such applications. Our work is a step toward incorporating practical points-to and side-effect analyses in realistic compilers.

Points-to Analysis and Side-Effect Analysis. Modification side-effect analysis (MOD) determines, for each statement, the variables whose values may be modified by that statement. The complementary USE analysis computes similar information for the uses of variable values. Such information plays a fundamental role in optimizing compilers and software tools: it enables a variety of other analyses (e.g., reaching definitions analysis, live variables analysis, etc.), which in turn are needed for code optimization and for program understanding, restructuring and testing. For brevity, we only discuss MOD analysis; all results trivially apply to USE analysis, because the two analysis problems are essentially identical.

Side-effect analysis for languages like C is difficult because of pointer usage; typically, a *points-to analysis* is needed to resolve pointer dereferences. Our work is focused on flow- and context-insensitive points-to analyses [11,18,21,17,5], which ignore the flow of control and the calling contexts of procedures. Such analyses are very efficient and can be used in production-strength compilers with little additional cost. Several combinations of a MOD analysis and a flow- and context-insensitive points-to analysis have been investigated [16,15,10]. Similarly to [16,10], we consider a MOD analysis based on Andersen’s points-to analysis for C [1]. Even though we investigate these specific analyses, our results also apply to similar flow- and context-insensitive points-to analyses (e.g., [18,17,5]) and related MOD analyses.

1.1 Separate Analysis of Modules

Optimizing compilers cannot use whole-program analyses for programs built with precompiled library modules. When compiling and optimizing a library module, the compiler has no available information about client modules. When compiling and optimizing a client module, only the library binary is available. Therefore, the compiler must use *separate analyses* that allow each module to be analyzed separately from the rest of the program. We define such separate analyses based on Andersen’s analysis and the corresponding MOD analysis.

Worst-Case Separate Analysis of Library Modules. The compilation and optimization of the library module is performed independently of any client modules. In this case, the separate analyses used by the optimizing compiler must make *worst-case assumptions* about the rest of the program. The resulting analysis solutions should represent all possible points-to and MOD relationships in the

¹ The work in [12] discusses programs built with more than one library module.

library module; these solutions can then be used for library compilation and optimization. In our approach, the worst-case separate analyses are implemented by adding *auxiliary statements* that model the effects of all possible statements in client modules. The library is combined with the auxiliary statements and the standard whole-program analyses are applied to it. This approach is easy to implement by reusing existing implementations of the whole-program analyses.

Summary-Based Separate Analysis of Client Modules. During the compilation and optimization of a client module, the compiler can employ separate analyses that use precomputed *summary information* about the called library module. Such summary-based analyses of the client module can compute more precise points-to and MOD information, as opposed to making worst-case assumptions about the library in the case when only the library binary is available. This improvement is important because the precision of the analyses has significant impact on subsequent analyses and optimizations [16,10].

The summary information should encode the effects of the library module on arbitrary client modules. Such information can be constructed at the time when the library module is compiled, and can be stored together with the library binary. The library summary can later be used during the separate analysis of any client module, and can be reused across different client modules. In our summary-based analyses, the client module is combined with the summary information, and the result is analyzed as if it were a complete program. This approach can reuse existing implementations of the corresponding whole-program analyses, which allows minimal implementation effort.

Summary Information. Our summaries contain a set of *summary elements*, where each element represents the points-to or MOD effects of one or more library statements. We extract an initial set of elements from the library code and then optimize it by merging equivalent elements and by eliminating irrelevant elements. The resulting summary is *precision-preserving*: with respect to client modules, the summary-based solutions are the same as the solutions that would have been computed if the standard whole-program analyses were possible.

Our approach for summary construction has several advantages. The summary can be generated completely independently of any callers and callees of the library; for example, unlike previous work, our approach can handle callbacks from the library to client modules. The summary construction algorithm is simple to implement, which makes it a good candidate for inclusion in realistic compilers. The summary optimizations reduce the cost of the summary-based analyses without sacrificing any precision. Finally, our experiments show that the cost of constructing and storing the summary is practical.

Contributions. The contributions of our work are the following:

- We propose an approach for separate points-to and MOD analyses of library modules, by using auxiliary statements to encode worst-case assumptions about the rest of the program. The approach can be easily implemented with existing implementations of the corresponding whole-program analyses.

- We present an approach for constructing summary information for a library module. The library summary is constructed independently of the rest of the program, and is optimized by merging equivalent summary elements and by eliminating irrelevant summary elements. With this summary, the summary-based analyses are as precise as the standard whole-program analyses.
- We show how to use the precomputed library summaries to perform separate points-to analysis and MOD analysis of client modules. The summary-based analyses can be implemented with minimal effort by reusing existing implementations of the corresponding whole-program analyses.
- We present empirical results showing that the worst-case points-to analysis and the summary construction algorithm are practical even for large library modules. The results also show that the summary optimizations can significantly reduce the size of the summary and the cost of the summary-based points-to analysis.

2 Whole-Program Analyses

In this section we present a conceptual model of Andersen’s points-to analysis for C [1] and a MOD analysis based on it. These whole-program analyses are the foundation for our worst-case and summary-based separate analyses.

We assume a simplified C-like program representation (defined formally in [12]), similar to the intermediate representations used in optimizing compilers. Similarly to [18,17,16,6,5,10], structures and arrays are represented as monolithic objects without distinguishing their individual elements. Calls to `malloc` and similar functions are replaced by statements of the form “ $x = \&heap_i$ ”, where $heap_i$ is a variable unique to the allocation site. Because of the weak type system of C (due to typecasting and union types), we assume that type information is not used in the points-to and MOD analyses [3], and therefore the representation is untyped. Figure 1 shows an example of a program with two modules.

Let V be the set of variables in the program representation. We classify the elements of V as (i) *global variables*, (ii) *procedure variables*, which denote the names of procedures, (iii) *local variables*, including formals, and (iv) *heap variables* introduced at heap-allocation sites. Andersen’s analysis constructs a *points-to graph* in which nodes correspond to variables from V . A directed edge (v_1, v_2) shows that v_1 may contain the address of v_2 . Each statement defines a *transfer function* that adds edges to points-to graphs. For example, the function for “ $*p = q$ ” is $f(G) = G \cup \{(x, y) \mid (p, x) \in G \wedge (q, y) \in G\}$. Conceptually, the analysis starts with an empty graph and applies transfer functions until a fixed point is reached. Figure 1 shows the points-to solution for the sample program.

We define a MOD algorithm which computes a set of modified variables $Mod(s) \subseteq V$ for each statement s . The algorithm is derived from similar MOD algorithms [16,15,10] by adding two *variable filters* that compensate for the some

² This analysis approach is the simplest to implement and therefore most likely to be the first one employed by realistic compilers. Our techniques can be easily adapted to approaches that use some form of type information.

input $Stmt$: set of statements $Proc$: set of procedures
 $SynMod: Stmt \rightarrow V \times \{D, I\}$ $Pt: V \rightarrow \mathcal{P}(V)$
 $Called: Stmt \rightarrow \mathcal{P}(Proc)$
output $Mod: Stmt \rightarrow \mathcal{P}(V)$
declare $ProcMod: Proc \rightarrow \mathcal{P}(V)$

```

[1]  foreach  $s \in Stmt$  do
[2]      if  $SynMod(s) = (v, D)$  then
[3]           $Mod(s) := \{v\}$ 
[4]          if  $v$  is global or static local then
[5]              add  $\{v\}$  to  $ProcMod(EnclosingProc(s))$ 
[6]      if  $SynMod(s) = (v, I)$  then
[7]           $Mod(s) := \{x \mid x \in Pt(v) \wedge active(s, x)\}$ 
[8]          add  $Mod(s)$  to  $ProcMod(EnclosingProc(s))$ 
[9]  while changes occur in  $Mod$  or  $ProcMod$  do
[10]     foreach call statement  $s \in Stmt$  do
[11]         foreach  $P \in Called(s)$  do
[12]              $Mod(s) := Mod(s) \cup \{x \mid x \in ProcMod(P) \wedge accessible(s, x)\}$ 
[13]         add  $Mod(s)$  to  $ProcMod(EnclosingProc(s))$ 

```

Fig. 2. Whole-program MOD analysis. $\mathcal{P}(X)$ denotes the power set of X .

parameter of s , or (iii) $v \in Pt(w)$ and $accessible(s, w)$ holds for some $w \in V$. For example, variable u from Figure 1 is active for all statements in procedures *exec*, *div*, and *neg*. With respect to calls 16 and 18, u is accessible because it is pointed to by actual g ; however, u is not accessible for call 14.

The MOD algorithm is shown in Figure 2. *SynMod* stores *syntactic modifications*, which are pairs (v, d) where v is a variable that occurs on the left-hand side of an assignment or a call, and $d \in \{D, I\}$ indicates whether the modification is direct or indirect. For example, in Figure 1, $SynMod(s)$ is (x, D) for statement 1 and (a, I) for statement 10. *Called* contains the procedures invoked by each call statement; indirect calls are resolved using the points-to solution. *ProcMod* stores the sets of variables modified by each procedure. Filters *active* and *accessible* are used whenever a variable is added to a *Mod* set (lines 7 and 12). In addition, we filter out *direct* modifications of non-static locals (lines 4–5), because the lifetime of the modified memory location terminates when the procedure returns. This filtering improves the precision of the analysis in the presence of recursion [12] and results in more compact summaries, as discussed in Section 4. Figure 1 shows some of the *Mod* sets for the sample program.

3 Worst-Case Separate Analysis of Library Modules

During the compilation and optimization of a library module, a compiler must use separate analyses that make worst-case assumptions about possible client modules. In our approach, these assumptions are introduced by adding *auxiliary statements* to the library. The combination of the auxiliary statements and the library is treated as a complete program and the whole-program analyses

```

global v_ph
proc p_ph( $f_1, \dots, f_n$ ) returns p_ph_ret {
  v_ph =  $f_i$  ( $1 \leq i \leq n$ )      v_ph = & $v$  ( $v \in V_{exp}$ )      v_ph = & $v\_ph$ 
  v_ph = *v_ph                    *v_ph = v_ph              v_ph = &p_ph
  v_ph = (*v_ph)(v_ph, ..., v_ph) (with  $m$  actuals)
  p_ph_ret = v_ph
}

```

Fig. 3. Placeholder procedure and auxiliary statements.

from Section 2 are applied to it. The resulting solutions are *safe abstractions* of all points-to and MOD relationships in all possible complete programs. These solutions can then be used for library compilation and optimization.

Given a library module *Lib*, consider a complete program p containing *Lib* and some client module. Let $PROG(Lib)$ be the (infinite) set of all such complete programs. We use V_p to denote the variable set of any such p . Let $V_L \subseteq V_p$ be the set of all variables that occur in statements in *Lib* (this set is independent of any particular p). Also, let $V_{exp} \subseteq V_L$ be the set of all variables that may be explicitly referenced by client modules; we refer to such variables as *exported*. Exported variables are either globals that could be directly accessed by library clients, or names of procedures that could be directly called by client code.

Example. We use module *Lib* from Figure 1 as our running example; for convenience, the module is shown again in Figure 4. For this module, $V_L = \{exec, p, fp, s, u, t, g, q, neg, r, i, j\}$. For the purpose of this example, we assume that $V_{exp} = \{g, exec\}$. Note that the complete program in Figure 1 is one of the (infinitely many) elements of $PROG(Lib)$.

Auxiliary Statements. The statements are located in a *placeholder procedure* p_ph which represents all procedures in all possible client modules. The statements use a *placeholder variable* v_ph which represents all global, local, and heap variables $v \in (V_p - V_L)$ for all $p \in PROG(Lib)$. The placeholder procedure and the auxiliary statements are shown in Figure 3. Each statement represents different kinds of statements that could occur in client modules; for example, “ $v_ph = *v_ph$ ” represents statements of the form “ $u = *w$ ”, where $u, w \in (V_p - V_L)$.

The indirect call through v_ph represents all calls originating from client modules. In the worst-case analyses, the targets of this call could be (i) p_ph , (ii) the procedures from V_{exp} , or (iii) any library procedure whose address is taken somewhere in *Lib*. To model all possible formal-actual pairs, the number of actuals m should be equal to the maximum number of formals for all possible target procedures. Similarly, all callbacks from the library are represented by indirect calls to p_ph ; thus, the number of formals n in p_ph should be equal to the maximum number of actuals used at indirect calls in the library.

Worst-Case Analyses. The worst-case points-to and MOD analyses combine the library module with the auxiliary statements and apply the whole-program analyses from Section 2. An important advantage of this approach is its simple implementation by reusing already existing implementations of the whole-program

```

global g
proc exec(p,fp) {
  local s,u,q,t
  13: t = p
  16: neg(q)
  19: *t = u
}
  11: s = 3
  14: (*fp)(g,t)
  17: q = &s
  20: g = t
}
  12: u = 4
  15: q = &u
  18: neg(q)
}

proc neg(r) {
  local i,j
  21: i = *r
  22: j = -i
  23: *r = j
}

 $Pt_{wc}(v) = \emptyset$  for  $v \in \{s, u, i, j, neg\}$ 
 $Pt_{wc}(v) = \{s, u\}$  for  $v \in \{q, r\}$ 
 $Pt_{wc}(v) = \{v\_ph, p\_ph, g, exec\}$  for every
other  $v \in V_L \cup \{v\_ph, p\_ph, f_i, p\_ph\_ret\}$ 

 $Mod_{wc}(s) = \{v\_ph, g\}$  for  $s \in \{14, 19\}$ 
and for the indirect call through  $v\_ph$ 
 $Mod_{wc}(23) = \{s, u\}$ 
 $Called_{wc}(s) = \{p\_ph, exec\}$  for  $s = 14$ 
and for the indirect call through  $v\_ph$ 

```

Fig. 4. Module *Lib* ($V_{exp} = \{g, exec\}$) and the corresponding worst-case solutions.

analyses. It can be proven that the resulting worst-case solution is a safe abstraction of all points-to and MOD relationships in all $p \in PROG(Lib)$ [12].

Example. Consider module *Lib* from Figure 4. For the purpose of this example assume that $V_{exp} = \{g, exec\}$. The library has one indirect call with two actuals; thus, p_ph should have two formals ($n = 2$). The indirect call through v_ph has possible targets *exec* and p_ph , and should have two actuals ($m = 2$). The points-to solution shown in Figure 4 represents all possible points-to pairs in all complete programs. For example, pair (p, v_ph) shows that p can point to some unknown variable from some client module; similarly, (p, p_ph) indicates that p can point to an unknown procedure from some client module. The computed call graph represents all possible calls in all complete programs. For example, $Called_{wc}(14) = \{p_ph, exec\}$ represents the possible callback from *exec* to some client module and the possible recursive call of *exec*. Similarly, the computed MOD solution represents all possible *Mod* sets in all complete programs.

4 Summary Construction and Summary-Based Analysis

In this section we present our approach for constructing summary information for a library module, and show how to use the library summaries for separate summary-based analysis of client modules.

Some previous work on context-sensitive points-to analysis [9, 23] uses *complete summary functions* to encode the cumulative effects of all statements in a procedure P and in all procedures transitively called by P . This approach can be used to produce summary information for a library module, by computing and storing the summary functions for all exported procedures. However, this technique makes the implicit assumption that a called procedure can be analyzed either before, or together with its callers. This assumption can be easily violated—for example, when a library module calls another library module, there are no guarantees that any summary information will be available for the callee [12]. In the presence of callbacks, the library module may call client modules that do not even exist at the time of summary construction. For example, for

1. Variable summary

$Procedures = \{exec, neg\}$	$Locals(exec) = \{p, fp, s, u, q, t\}$
$Globals = \{g\}$	$Locals(neg) = \{r, i, j\}$
2. Points-to summary

$proc\ exec(p, fp)$	$q = \&s$	$*t = u$	$i = *r$
$t = p$	$neg(q)$	$g = t$	$*r = j$
$(*fp)(g, t)$	$q = \&u$	$proc\ neg(r)$	
3. Mod summary

$SynMod(exec) = \{(t, I), (g, D)\}$	$SynMod(neg) = \{(r, I)\}$
$SynCall(exec) = \{(fp, I), (neg, D)\}$	$SynCall(neg) = \emptyset$

Fig. 5. Basic summary for module *Lib*.

module *Lib* from Figure 4, the effects of *exec* cannot be expressed by a summary function because of the callback to some unknown client module.

We use a different summary construction approach that has several advantages. The summary can be constructed independently of any callers and callees of the library; therefore, our approach can handle callbacks. The summary construction algorithm is inexpensive and simple to implement, which makes it a good candidate for inclusion in realistic compilers. The summary is *precision-preserving*: for every statement in the client module, the MOD solution computed by the summary-based analyses is the same as the solution that would have been computed if the standard whole-program analyses were possible. This ensures the best possible cost and precision for the users of the MOD information.

The *basic summary* is the simplest summary information produced by our approach. Figure 5 shows the basic summary for module *Lib* from Figure 4. The summary has three parts. The *variable summary* contains all relevant library variables. The *points-to summary* contains all library statements that are relevant to points-to analysis. The *proc* declarations are used by the subsequent points-to analysis to model the formal-actual pairings at procedure calls. The *Mod summary* contains syntactic modifications and syntactic calls for each library procedure. A syntactic modification (defined in Section 2) is a pair (v, D) or (v, I) representing a direct or indirect modification. A syntactic call is a similar pair indicating a direct or indirect call through v . The Mod summary does not include direct modifications of non-static local variables—as discussed in Section 2, such modifications can be filtered out by the MOD analysis.

In the summary-based separate analysis, the program representation of a client module is combined with the library summary and the result is analyzed as if it were a complete program. Thus, already existing implementations of the whole-program analyses from Section 2 can be reused with only minor adjustments, which makes the approach simple to implement. Clearly, the computed points-to and MOD solutions are the same as the solutions that would have been produced by the standard whole-program analyses. However, the cost of the summary-based analyses is essentially the same as the cost of the whole-program analyses. The next section presents summary optimizations that reduce this cost.

1. Variable summary

$Procedures = \{exec, neg\}$	$Locals(exec) = \{fp, s, u\}$	$Reps = \{rep_1, rep_2\}$
$Globals = \{g\}$	$Locals(neg) = \{i, j\}$	
2. Points-to summary

<code>proc exec(rep1,fp)</code>	<code>rep2=&s</code>	<code>*rep1=u</code>	<code>i=*rep2</code>
<code>(*fp)(g,rep1)</code>	<code>rep2=&u</code>	<code>g=rep1</code>	<code>*rep2=j</code>
3. Mod summary

$SynMod(exec) = \{(rep_1, I), (g, D)\}$	$SynMod(neg) = \{(rep_2, I)\}$
$SynCall(exec) = \{(fp, I), (neg, D)\}$	$SynCall(neg) = \emptyset$

Fig. 6. Optimization through variable substitution.

5 Summary Optimizations

In this section we describe three techniques for optimizing the basic summary. The resulting *optimized summary* has two important features. First, the summary is precision-preserving: for each statement in a client module, the summary-based MOD solution computed with the optimized summary is the same as the solution computed with the basic summary. Second, as shown by our experiments in Section 6 the cost of the summary-based analyses is significantly reduced when using the optimized summary, compared to using the basic summary.

Variable Substitution. Variable substitution is a technique for reducing the cost of points-to analysis by replacing a set of variables with a single representative variable. We use a specific precision-preserving substitution proposed in [11, 13]. With this technique we produce a more compact summary, which in turn reduces the cost of the subsequent summary-based analyses without any loss of precision.

Two variables are *equivalent* if they have the same points-to sets. The substitution is based on mutually disjoint sets of variables V_1, \dots, V_k such that for each set V_i (i) all elements of V_i are equivalent, (ii) no element of V_i has its address taken (i.e., no element is pointed to by other variables), and (iii) no element of V_i is exported. For example, for module *Lib* in Figure 4 possible sets are $V_1 = \{p, t\}$ and $V_2 = \{q, r\}$. Each V_i has associated a *representative variable* rep_i . We optimize the points-to summary by replacing all occurrences of a variable $v \in V_i$ with rep_i . In addition, in the Mod summary, every pair (v, I) is replaced with (rep_i, I) . It can be proven that this optimization is precision-preserving—given the modified summary, the subsequent MOD analysis computes the same solution as the solution computed with the basic summary.

We identify equivalent variables using a linear-time algorithm presented in [13], which extends a similar algorithm from [11]. The algorithm constructs a *subset graph* in which nodes represent expressions and edges represent subset relationships between the points-to solutions for the nodes. For example, edge (q, p) shows that $Pt(q) \subseteq Pt(p)$. The graph represents all subset relationships that can be directly inferred from individual library statements. A strongly-connected component (SCC) in the graph corresponds to expressions with equal points-to sets. The algorithm constructs the SCC-DAG condensation of the graph and

traverses it in topological sort order; this traversal identifies SCCs with equal points-to sets. Due to space limitations, the details of the algorithm are not presented in this paper; we refer the interested reader to [13] for more details.

Example. Consider module *Lib* from Figure 4. Since t is assigned the value of p and there are no indirect assignments to t (because its address is not taken), t has exactly the same points-to set as p . In this case, the algorithm can identify set $V_1 = \{p, t\}$. Similarly, the algorithm can detect set $V_2 = \{q, r\}$. After the substitution, the summary can be simplified by eliminating trivial statements. For example, “ $t=p$ ” is transformed into “ $\text{rep1}=\text{rep1}$ ”, which can be eliminated. Call “ $\text{neg}(\text{rep2})$ ” can also be eliminated. Since this is the only call to *neg* (and *neg* is not exported), declaration “ $\text{proc neg}(\text{rep2})$ ” can be removed as well. Figure 6 shows the resulting summary, derived from the summary in Figure 5.

In addition to producing a more compact summary, we use the substitution to reduce the cost of the worst-case points-to analysis. During the analysis, every occurrence of $v \in V_i$ is treated as an occurrence of rep_i ; in the final solution, the points-to set of v is defined to be the same as the points-to set computed for rep_i . This technique produces the same points-to sets as the original analysis.

Statement Elimination. After variable substitution, the points-to summary is simplified by removing statements that have no effect on client modules. A variable $v \in V_L$ is *client-inactive* if v is a non-static local in procedure P and there is no path from P to procedure p_ph in the call graph used for the worst-case MOD analysis of the library module. The worst-case call graph represents all possible call graphs for all complete programs; thus, for any such v , $\text{active}(s, v)$ (defined in Section 2) is false for all statements s in all client modules.

Let $\text{Reach}_{wc}(u)$ be the set of all variables reachable from u in the points-to graph computed by the worst-case points-to analysis of the library module.³ A variable $v \in V_L$ is *client-inaccessible* if (i) v is not a global, static local, or procedure variable, and (ii) v does not belong to $\text{Reach}_{wc}(u)$ for any global or static local variable u , including v_ph . It is easy to show that in this case $\text{accessible}(s, v)$ is false for all call statements s in all client modules.

In the summary-based MOD analysis, any variable that is client-inactive or client-inaccessible will not be included in any *Mod* set for any statement in a client module. We refer to such variables as *removable*. The optimization eliminates from the points-to summary certain statements that are only relevant with respect to removable variables. As a result, the summary-based points-to analysis computes a solution in which some points-to pairs (p, v) are missing. For any such pair, v is a removable variable; thus, the optimization is precision-preserving (i.e., *Mod* sets for statements in client modules do not change).

The optimization is based on the fact that certain variables can only be used to access removable variables. Variable v is *irrelevant* if $\text{Reach}_{wc}(v)$ contains only removable variables. Certain statements involving irrelevant variables can be safely eliminated from the points-to summary: (i) “ $p = \&q$ ”, if q is removable and irrelevant, (ii) “ $p = q$ ”, “ $p = *q$ ”, and “ $*p = q$ ”, if p or q is irrelevant, and

³ w is reachable from u if there exists a path from u to w containing at least one edge.

1. Variable summary

$$\begin{aligned} \text{Procedures} &= \{exec, neg\} & \text{Locals}(exec) &= \{fp\} & \text{Reps} &= \{rep_1\} \\ \text{Globals} &= \{g\} & \text{Locals}(neg) &= \emptyset \end{aligned}$$
2. Points-to summary

$$\text{proc } exec(rep_1, fp) \quad (*fp)(g, rep_1) \quad g=rep_1$$
3. Mod summary

$$\begin{aligned} \text{SynMod}(exec) &= \{(rep_1, I), (g, D)\} & \text{SynMod}(neg) &= \emptyset \\ \text{SynCall}(exec) &= \{(fp, I), (neg, D)\} & \text{SynCall}(neg) &= \emptyset \end{aligned}$$

Fig. 7. Final optimized summary.

(iii) calls “ $p = f(q_1, \dots, q_n)$ ” and “ $p = (*fp)(q_1, \dots, q_n)$ ”, if all of p, q_1, \dots, q_n are irrelevant. Intuitively, the removal of such statements does not “break” points-to chains that end at non-removable variables. It can be proven that the points-to solution computed after this elimination differs from the original solution only by points-to pairs (p, v) in which v is a removable variable [12].⁴

Example. Consider module *Lib* and the worst-case solutions from Figure 4. Variables $\{r, i, j\}$ are client-inactive because the worst-case call graph does not contain a path from *neg* to *p_ph*. Variables $\{p, fp, s, u, q, t, r, i, j\}$ are client-inaccessible because they are not reachable from *g* or *v_ph* in the worst-case points-to graph. Variables $\{s, u, i, j\}$ have empty points-to sets and are irrelevant. Variables $\{q, r, rep_2\}$ can only reach *s* and *u* and are also irrelevant. Therefore, the following statements can be removed from the points-to summary in Figure 6: `rep2=&s`, `rep2=&u`, `*rep1=u`, `i=*rep2`, and `*rep2=j`.

Modification Elimination. This optimization removes from the Mod summary each syntactic modification (v, I) for which $Pt_{wc}(v)$ contains only removable variables. Clearly, this does not affect the *Mod* sets of statements in client modules. In Figure 6, (rep_2, I) can be removed because rep_2 points only to removable variables *s* and *u*. The final optimized summary for our running example is shown in Figure 7.

6 Empirical Results

For our initial experiments, we implemented the worst-case and summary-based points-to analyses, as well as the summary construction techniques from Section 5. Our implementation of Andersen’s analysis is based on the BANE toolkit for constraint-based analysis [6]; the analysis is performed by generating and solving a system of set-inclusion constraints. We measured (i) the cost of the worst-case points-to analysis of the library, (ii) the cost of constructing the optimized summary, (iii) the size of the optimized summary, and (iv) the cost of

⁴ Identifying client-inaccessible and irrelevant variables requires reachability computations in the worst-case points-to graph; the cost of these traversals can be reduced by merging *v_ph* with all of its successor nodes, without any loss of precision [12].

Table 1. Data programs and libraries. Last two columns show absolute and relative library size in lines of code and in number of pointer-related statements.

Program	Library	LOC	Statements
gnuplot-3.7.1	libgd-1.3	22.2K (34%)	2965 (6%)
gasp-1.2	libiberty	11.2K (43%)	6259 (50%)
bzip2-0.9.0c	libbz2	4.5K (71%)	7263 (86%)
unzip-5.40	zlib-1.1.3	8.0K (29%)	9005 (33%)
fudgit-2.41	readline-2.0	14.8K (50%)	10788 (39%)
cjpeg-5b	libjpeg-5b	19.1K (84%)	17343 (84%)
tiff2ps-3.4	libtiff-3.4	19.6K (94%)	20688 (84%)
povray-3.1	libpng-1.0.3	25.7K (19%)	25322 (23%)

the summary-based points-to analysis of the client module. At present, our implementation does not perform MOD analysis. Nevertheless, these preliminary results are important because the total cost of the two analyses is typically dominated by the cost of the points-to analysis [16,15].

Table 1 describes our C data programs. Each program contains a well-defined library module, which is designed as a general-purpose library and is developed independently of any client applications. For example, `unzip` is an extraction utility for compressed archives which uses the general-purpose data compression library `zlib`. We added to each library module a set of stubs representing the effects of standard library functions (e.g., `strcpy`, `cos`, `rand`); the stubs are summaries produced by hand from the specifications of the library functions [5].

Table 1 shows the number of lines of source code for each library, as an absolute value and as percentage of the number for the whole program. For example, for `povray` 19% (25.7K) of the source code lines are in the library module, and the remaining 81% (108.2K) are in the client module. The table also shows the number of pointer-related statements in the intermediate representation of the library, as an absolute value and as percentage of the whole-program number. These numbers represent the size of the input for the points-to analysis.

For our first set of experiments, we measured the cost of the worst-case points-to analysis and the cost of constructing the optimized summary. The results are shown in Figure 8(a).⁶ Column T_{wc} shows the running time of the worst-case points-to analysis of the library module. Column T_{sub} contains the time to compute the variable substitution. Column T_{elim} shows the time to identify removable and irrelevant variables in order to perform statement elimination and modification elimination. Finally, column S_{max} contains the maximum amount of memory needed during the points-to analysis and the summary construction.

⁵ We plan to investigate how our approach can be used to produce summaries for the standard libraries. This problem presents interesting challenges, because many of the standard libraries operate in the domain of the operating system.

⁶ All experiments were performed on a 360MHz *Sun Ultra-60* with 512Mb physical memory. The reported times are the median values out of five runs.

(a) Library Module					(b) Client Module				
Library	T_{wc} (s)	T_{sub} (s)	T_{elim} (s)	S_{max} (Mb)	Program	T_b (s)	Δ_T	S_b (Mb)	Δ_S
libgd	3.9	0.4	0.1	10.1	gnuplot	47.3	4%	79.6	5%
libiberty	5.8	0.5	0.1	10.4	povray	111.6	17%	146.7	16%
libbz2	4.1	0.8	0.1	8.5	unzip	21.0	25%	39.5	16%
zlib	6.5	1.0	0.1	11.4	fudgit	39.8	21%	59.5	17%
readline	10.2	1.3	0.1	12.3	gasp	9.7	32%	18.6	26%
libjpeg	15.1	2.5	0.1	19.7	cjpeg	15.7	59%	32.1	56%
libtiff	23.4	3.6	0.2	25.2	tiff2ps	22.5	61%	37.5	59%
libpng	19.8	4.0	0.2	21.8	bzip2	5.4	69%	13.0	54%

Fig. 8. (a) Cost of the library analyses: running time (in seconds) of the worst-case points-to analysis (T_{wc}) and time for constructing the optimized summary (T_{sub} and T_{elim}). S_{max} is the maximum memory usage. (b) Cost of the points-to analyses of the client. T_b is the analysis time with the basic summary, and Δ_T is the reduction in analysis time when using the optimized summary. S_b and Δ_S are the corresponding measurements for analysis memory.

Clearly, the cost of the worst-case points-to analysis and the cost of the summary construction are low. Even for the larger libraries (around 20K LOC), the running time and memory usage are practical. These results indicate that both the worst-case points-to analysis and the summary construction algorithm are realistic candidates for inclusion in optimizing compilers.

Our second set of experiments investigated the difference between the basic summary from Section 4 and the optimized summary from Section 5. We first compared the sizes of the two summaries. For brevity, in this paper we summarize these measurements without explicitly showing the summary sizes. The size of the optimized summary was between 21% and 53% (31% on average) of the size of the basic summary. Clearly, the optimizations described in Section 5 result in significant compaction in the library summaries. In addition, we measured the size of the optimized summary as percentage of the size of the library binary. This percentage was between 14% and 76% (43% on average), which shows that the space overhead of storing the summary is practical.⁷

Next, we measured the differences between the basic summary and the optimized summary with respect to the cost of the summary-based points-to analysis of the client module. The results are shown in Figure 8(b). The order of the programs in the table is based on the relative size of the library, as shown by the percentages in the last column of Table 1. Column T_b shows the analysis time when using the basic summary. Column Δ_T shows the reduction in analysis time when using the optimized summary. The reduction is proportional to the relative size of the library. For example, in `bzip2` the majority of the program is in the library, and the cost reduction is significant. In `gnuplot` only 6% of the pointer-related statements are in the library, and the analysis cost is reduced

⁷ The sizes depend on the file format used to store the summaries. We use a simple text-based format; more optimized formats could further reduce summary size.

accordingly. Similarly, the reduction Δ_S in the memory usage of the analysis is proportional to the relative size of the library.

The results from these experiments clearly show that the optimizations from Section 5 can have significant beneficial impact on the size of the summary and the cost of the subsequent summary-based points-to analysis.

7 Related Work

The work in [16,10] investigates various points-to analyses and their applications, including Andersen’s analysis and MOD analyses based on it. Our conceptual whole-program MOD analysis is based on this work. At call sites, [10] filters out certain variables whose lifetime has terminated; our use of the *active* filter is similar to this approach.

A general approach for analyzing program fragments is presented in [14]. The interactions of the fragment with the rest of the program are modeled by summary values and summary functions. The worst-case separate analyses are examples of fragment analyses, in which auxiliary statements play the same role as summary values and summary functions. Unlike [14], we use abstractions not only for lattice elements (i.e., points-to graphs and *Mod* sets), but also for statements, procedures, and call graphs.

The summary-based separate analyses are also examples of fragment analyses, in which the summary information is used similarly to the summary functions from [14]. However, instead of using a complete summary function for each exported library procedure, we use a set of elementary transfer functions. This approach is different from summary construction techniques based on context-sensitive analysis [9,23]. In these techniques, a called procedure is analyzed before or together with its callers. In contrast, our summaries can be constructed completely independently from the rest of the program. This allows handling of callbacks to unknown client modules and calls to unanalyzed library modules. In addition, for compilers that already incorporate a flow- and context-insensitive points-to analysis, our summary construction approach is significantly easier to implement than the context-sensitive techniques for summary construction. This minimal implementation effort is an important advantage for realistic compilers.

Escape analysis for Java determines if an object can escape the method or thread that created it. Our notion of accessible variables is similar to the idea of escaping objects. Some escape analyses [20,4] calculate specialized points-to information as part of the analysis. In this context, escape summary information for a module can be computed in isolation from the rest of the program. The techniques used in this manner for escape analysis cannot be used for general-purpose points-to analysis.

Flanagan and Felleisen [7] present an approach for componential set-based analysis of functional languages. They derive a simplified constraint system for each program module and store it in a constraint file; these systems are later combined and information is propagated between them. The constraint files used in this work are similar in concept to our use of library summaries.

Guyer and Lin [8] propose annotations for describing libraries in the domain of high-performance computing. The annotations encode high-level semantic information and are produced by a library expert. Some annotations describe the points-to and MOD/USE effects of library procedures, in a manner resembling complete summary functions. The annotations are used for source-to-source optimizations of the library and application code. This approach allows domain experts to produce high-level information that cannot be obtained through static analysis. However, such approaches cannot be used in general-purpose compilers.

Sweeney and Tip [19] describe analyses and optimizations for the removal of unused functionality in Java modules. Worst-case assumptions are used for code located outside of the optimized modules. This work also presents techniques that allow a library creator to specify certain library properties (e.g., usage of reflection) that are later used when optimizing the library and its clients.

8 Conclusions and Future Work

Traditional whole-program analyses cannot be used in the context of a modular development process. This problem presents a serious challenge for the designers of program analyses. In this paper we show how Andersen’s points-to analysis and the corresponding MOD analysis can be used for programs built with pre-compiled libraries. Our approach can be trivially extended to USE analysis. In addition, our techniques can be applied to other flow- and context-insensitive points-to analyses (e.g., [18, 17, 5]) and to MOD/USE analyses based on them.

We show how to perform worst-case analysis of library modules and summary-based analysis of client modules. These separate analyses can reuse already existing implementations of the corresponding whole-program analyses. We also present an approach for constructing summary information for library modules. The summaries can be constructed completely independently from the rest of the program; unlike previous work, this approach can handle callbacks to unknown clients and calls to unanalyzed libraries. Summary construction is inexpensive and simple to implement, which makes it a practical candidate for inclusion in optimizing compilers. We present summary optimizations that can significantly reduce the cost of the summary-based analyses without sacrificing any precision; these savings occur every time a new client module is analyzed.

An interesting direction of future research is to investigate separate analyses derived from flow-insensitive, context-sensitive points-to analyses. In particular, it is interesting to consider what kind of precision-preserving summary information is appropriate for such analyses. Another open problem is to investigate separate analyses and summary construction in the context of standard analyses that need MOD/USE information (e.g., live variables analysis and reaching definitions analysis), especially when the target analyses are flow-sensitive.

Acknowledgments. We thank Matthew Arnold for his comments on an earlier version of this paper. We also thank the reviewers for their helpful suggestions for improving the paper. This research was supported, in part, by NSF grants CCR-9804065 and CCR-9900988, and by Siemens Corporate Research.

References

1. L. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.
2. R. Chatterjee, B. G. Ryder, and W. Landi. Relevant context inference. In *Symposium on Principles of Programming Languages*, pages 133–146, 1999.
3. B. Cheng and W. Hwu. Modular interprocedural pointer analysis using access paths. In *Conference on Programming Language Design and Implementation*, pages 57–69, 2000.
4. J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 1–19, 1999.
5. M. Das. Unification-based pointer analysis with directional assignments. In *Conference on Programming Language Design and Implementation*, pages 35–46, 2000.
6. M. Fähndrich, J. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Conference on Programming Language Design and Implementation*, pages 85–96, 1998.
7. C. Flanagan and M. Felleisen. Componential set-based analysis. *ACM Trans. Programming Languages and Systems*, 21(2):370–416, Mar. 1999.
8. S. Guyer and C. Lin. Optimizing the use of high performance software libraries. In *Workshop on Languages and Compilers for Parallel Computing*, 2000.
9. M. J. Harrold and G. Rothermel. Separate computation of alias information for reuse. *IEEE Trans. Software Engineering*, 22(7):442–460, July 1996.
10. M. Hind and A. Pioli. Which pointer analysis should I use? In *International Symposium on Software Testing and Analysis*, pages 113–123, 2000.
11. A. Rountev and S. Chandra. Off-line variable substitution for scaling points-to analysis. In *Conference on Programming Language Design and Implementation*, pages 47–56, 2000.
12. A. Rountev and B. G. Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. Technical Report 423, Rutgers University, Oct. 2000.
13. A. Rountev and B. G. Ryder. Practical points-to analysis for programs built with libraries. Technical Report 410, Rutgers University, Feb. 2000.
14. A. Rountev, B. G. Ryder, and W. Landi. Data-flow analysis of program fragments. In *Symposium on the Foundations of Software Engineering*, LNCS 1687, 1999.
15. B. G. Ryder, W. Landi, P. Stocks, S. Zhang, and R. Altucher. A schema for interprocedural side effect analysis with pointer aliasing. Technical Report 336, Rutgers University, May 1998. To appear in ACM TOPLAS.
16. M. Shapiro and S. Horwitz. The effects of the precision of pointer analysis. In *Static Analysis Symposium*, LNCS 1302, pages 16–34, 1997.
17. M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Symposium on Principles of Programming Languages*, pages 1–14, 1997.
18. B. Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages*, pages 32–41, 1996.
19. P. Sweeney and F. Tip. Extracting library-based object-oriented applications. In *Symposium on the Foundations of Software Engineering*, pages 98–107, 2000.
20. J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 187–206, 1999.
21. S. Zhang, B. G. Ryder, and W. Landi. Program decomposition for pointer aliasing: A step towards practical analyses. In *Symposium on the Foundations of Software Engineering*, pages 81–92, 1996.

A Novel Probabilistic Data Flow Framework^{*}

Eduard Mehofer¹ and Bernhard Scholz²

¹ Institute for Software Science
University of Vienna, Austria
`mehofer@par.univie.ac.at`

² Institute of Computer Languages
Vienna University of Technology, Austria
`scholz@complang.tuwien.ac.at`

Abstract. Classical data flow analysis determines whether a data flow fact may hold or does not hold at some program point. Probabilistic data flow systems compute a range, i.e. a probability, with which a data flow fact will hold at some program point. In this paper we develop a novel, practicable framework for probabilistic data flow problems. In contrast to other approaches, we utilize execution history for calculating the probabilities of data flow facts. In this way we achieve significantly better results. Effectiveness and efficiency of our approach are shown by compiling and running the SPECint95 benchmark suite.

1 Introduction

Classical data flow analysis determines whether a data flow fact may hold or does not hold at some program point. For generating highly optimized code, however, it is often necessary to know the *probability* with which a data flow fact will hold during program execution (cf. [10,11]). In probabilistic data flow systems control flow graphs annotated with edge probabilities are employed to compute the probabilities of data flow facts. Usually, edge probabilities are determined by means of profile runs based on representative input data sets. These probabilities denote heavily and rarely executed branches and are used to weight data flow facts when propagating them through the control flow graph.

Consider the example shown in Fig. 1 to discuss classical and probabilistic data flow analysis. For the sake of simplicity we have chosen as data flow problem the reaching definitions problem [8]. The control flow graph G of our running example consists of two subsequent branching statements inside a loop and four definitions d_1 to d_4 . Variable X is defined at edges $2 \rightarrow 4$ and $5 \rightarrow 7$ by d_1 and d_3 . Similarly, variable Y is assigned a value at edges $3 \rightarrow 4$ and $6 \rightarrow 7$ by d_2 and d_4 . Classical reaching definition analysis yields that definitions d_1 to d_4 may reach nodes 1 to 8. The solution is a conservative approximation valid for all possible program runs. However, when we consider specific program runs, we

^{*} This research is partially supported by the Austrian Science Fund as part of Aurora Project “Languages and Compilers for Scientific Computation” under Contract SFB-011.

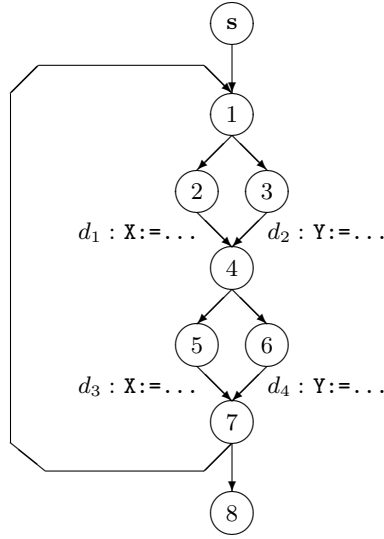


Fig. 1. Running example.

can compute a numerical value denoting the probability with which a definition actually may reach a node during execution. Ramalingam [13] presented a data flow framework which computes the probability of data flow facts, once every edge in the control flow graph has been annotated with a probability. In order to get an idea of the precision of his results, we defined in [9] the best solution \mathcal{S}_{best} that one can theoretically obtain and compared both. We showed that the differences between the theoretically best solution and Ramalingam's solution can be considerable and improvements are necessary. However, the computation of the theoretically best solution is too expensive in general and, hence, not feasible in practice. The modifications of the equation system described in [9] resulted in some improvements, but there is still potential for further improvements left.

Two reasons are responsible for the deviations between the theoretically best solution and Ramalingam's approach. On the one hand, program paths are reduced to edge probabilities. On the other hand, it is an execution history independent approach, i.e. it is assumed that particular branches are independent of execution history, which obviously is not true in reality. Since edge probabilities are indispensable to get an efficient handle on the problem, we focus on execution history in order to get better results. Consider a program run for our example in Fig. 1 that performs 10 iterations. At the beginning the left branches [1,2,4,5,7] are executed and in the last iteration the right branches [1,3,4,6,7,8] are taken. Without execution history edges $4 \rightarrow 5$ and $4 \rightarrow 6$ are dealt with independently of the incoming edges $2 \rightarrow 4$ and $3 \rightarrow 4$. However by correlating outgoing edges with incoming ones, it can be recognized that paths [2,4,6] and [3,4,5] are never taken. Hence, d_3 cannot reach node 5, since d_3 is killed on edge $2 \rightarrow 4$. Similarly,

it can be detected that d_4 cannot reach node **6**. Finally, since the loop is exited via path **[6,7,8]**, it can be determined that definition d_2 cannot reach node **8**.

In this paper we present a novel probabilistic data flow analysis framework (PDFA) which realizes an *execution history based approach*, i.e. the execution history is taken into account during the propagation of the probabilities through the control flow graph. Our approach is unique in utilizing execution history. We show that in this way significantly better results can be achieved with nearly the same computational effort as other approaches.

The paper is organized as follows. In Section **2** we describe the basic notions required to present our approach. In Section **3** we outline the basic ideas behind \mathcal{S}_{best} and Ramalingam’s history-independent approach and compare the results obtained by both approaches for our running example. Our novel approach is developed in Section **4**. In Section **5** we compare the probabilistic results of the individual approaches for the SPECint95 benchmark suite and present time measurements. Related work is surveyed in Section **6** and, finally, we draw our conclusions in Section **7**.

2 Preliminaries

Programs are represented by *directed flow graphs* $G = (N, E, \mathbf{s}, \mathbf{e})$, with node set N and edge set $E \subseteq N \times N$. Edges $m \rightarrow n \in E$ represent basic blocks of instructions and model the nondeterministic branching structure of G . *Start node* \mathbf{s} and *end node* \mathbf{e} are assumed to be free of incoming and outgoing edges, respectively. An element π of the set of paths Π of length k is a finite sequence $\pi = [n_1, n_2, \dots, n_k]$ with $k \geq 1$, $n_i \in N$ for $1 \leq i \leq k$ and for all $i \in \{1, \dots, k-1\}$, $n_i \rightarrow n_{i+1} \in E$.

A *program run* π_r is a path, which starts with node \mathbf{s} and ends in node \mathbf{e} . The set of all *immediate predecessors* of a node n is denoted by $\text{pred}(n) = \{m \mid (m, n) \in E\}$. Function **occurs** : $(N \cup \Pi) \times \Pi \rightarrow \mathbb{N}_0$ denotes the number of occurrences of a node/subpath in a path.

As usual, a monotone data flow analysis problem is a tuple $DFA = (L, \wedge, F, c, G, M)$, where L is a bounded semilattice with meet operation \wedge , $F \subseteq L \rightarrow L$ is a monotone function space associated with L , $c \in L$ are the “data flow facts” associated with start node \mathbf{s} , $G = (N, E, \mathbf{s}, \mathbf{e})$ is a control flow graph, and $M : E \rightarrow F$ is a map from G ’s edges to data flow functions.

For bitvector problems the semilattice L is a powerset 2^D of finite set D . An element χ in 2^D represents a function from D to $\{0, 1\}$. $\chi(d)$ is 1, if d is element of χ , 0 otherwise.

3 Abstract Run and Ramalingam's Approach

Abstract Run. The theoretical best solution \mathcal{S}_{best} can be determined by an abstract run [9]. The abstract run computes (1) the frequency $C(u)$ with which node u occurs in program run π_r and (2) the frequency $C(u, d)$ with which data fact d is true in node u for program run π_r . While $C(u)$ can be determined very easily, the computation of $C(u, d)$ is based on *monotone data flow problems* [8]: Whenever a node is reached, an associated function which describes the effect of that node on the data flow information is executed (for the details on computing $C(u)$ and $C(u, d)$ rf. to [9]).

Definition 1

$$\mathcal{S}_{best}(u, d) = \begin{cases} \frac{C(u, d)}{C(u)} & \text{if } C(u) \neq 0, \\ 0 & \text{otherwise.} \end{cases} \quad (1)$$

The definition above combines both frequencies. If frequency $C(u)$ of node u is zero, then $\mathcal{S}_{best}(u, d)$ is zero as well.

Table 1. Running example: Result of Abstract Run.

$\mathcal{S}_{best}(u, d)$	d_1	d_2	d_3	d_4	$C(u)$
s	0	0	0	0	1
1	0	0	0.9	0	10
2	0	0	0.889	0	9
3	0	0	1	0	1
4	0.9	0.1	0.1	0	10
5	1	0	0	0	9
6	0	1	1	0	1
7	0	0	1	0.1	10
8	0	0	1	1	1

Table 1 summarizes the results of our running example with program run π_r . Columns correspond to definitions (d_1 to d_4), and rows to nodes of the control flow graph. If $\mathcal{S}_{best}(u, d)$ is 0, definition d does not reach node u (impossible event). If $\mathcal{S}_{best}(u, d)$ is 1, definition d reaches node u each time (certain event). Any other numerical value in the range between 0 and 1 represents the probability for definition d reaching node u . E.g. consider $\mathcal{S}_{best}(4, d_1)$ and $\mathcal{S}_{best}(4, d_2)$. Node 4 occurs in program run π_r 10 times. Hence, the denominator of Equation 1 is 10 in both cases. To determine the number of times d_1 and d_2 reach node 4, we trace definitions d_1, d_2 in execution path π_r . Since edge $2 \rightarrow 4$ is executed 9 times, d_1 reaches node 4 at least 9 times. Definition d_1 is killed on edge $5 \rightarrow 7$ and in the last iteration node 4 is entered through edge $3 \rightarrow 4$. Therefore, definition d_1 does not hold true in node 4 for the last iteration. Hence, the number of times d_1 reaches node 4 is 9 and $\mathcal{S}_{best}(4, d_1) = 9/10$. Similarly, since edge $3 \rightarrow 4$ is

executed once and d_2 does not reach node 4 via edge $2 \rightarrow 4$, $\mathcal{S}_{best}(4, d_2) = 1/10$. Note that classical reaching definitions analysis yields that every definition can reach each node except the start node. Nevertheless, an abstract run is not a viable approach. The main drawback stems from the tremendous size of program path π_r and the resulting execution time.

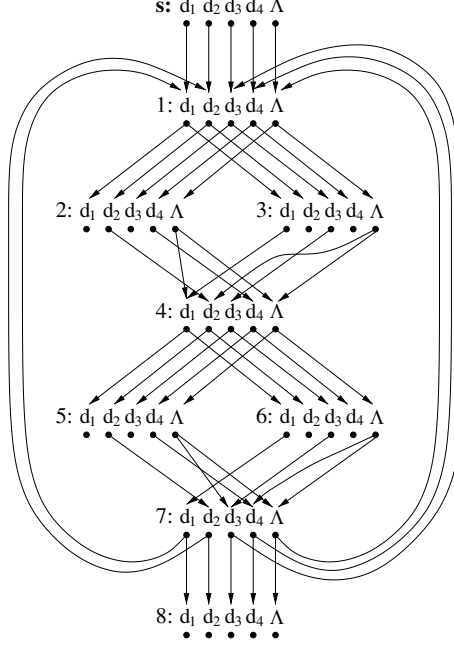


Fig. 2. Exploded flow graph of running example.

Ramalingam's Approach. Ramalingam [13] presents a framework for finite bi-distributive subset problems which estimates how often or with which probability a fact holds true during program execution. It is based on exploded control flow graphs introduced by Reps et al. [14] and Markov chains (with minor changes). Fig. 2 depicts the exploded control flow graph (ECFG) of our running example. The exploded control flow graph has $N \times D_\Lambda$ nodes, where D_Λ denotes data fact set D extended by symbol Λ . Edges of the extended control flow graph are derived from the representation relation (see [14]). Based on the ECFG, a linear equation system in \mathbb{R}^+ solves the expected frequencies which are used to compute the probabilities: $y(v, d)$ denotes the expected frequency for fact d to hold true at node v , and $y(v, \Lambda)$ gives the expected number of times that node v is executed. Thus $Prob(v, d) = y(v, d)/y(v, \Lambda)$ yields the probability for fact d to hold true in node v . The linear equation system is given as follows.

Ramalingam's Equation System:

$$\begin{aligned}
 &y(\mathbf{s}, A) = 1 \\
 &\text{for all } d \text{ in } D: \\
 &\quad y(\mathbf{s}, d) = c(d) \\
 &\text{for all } v \text{ in } N \setminus \{\mathbf{s}\}: \text{ for all } \delta \text{ in } D_A: \\
 &\quad y(v, \delta) = \sum_{(u, \delta') \in \mathbf{pred}(v, \delta)} p(u, v) * y(u, \delta')
 \end{aligned}$$

where $\mathbf{pred}(v, \delta)$ denotes the set of predecessors of node (v, δ) in the *ECFG* ($v \in N$, $\delta \in D_A$) and $p(u, v)$ denotes the probability that execution will follow edge $u \rightarrow v$ once u has been reached.

Table 2 lists the results of that approach for our running example. In comparison to Table 1, we can see deviations due to the reduction of the entire path to simple edge probabilities and the assumption that an incoming edge is independent from an outgoing edge. E.g. consider the probability of definition d_4 to reach node **6**. Although the abstract run yields probability 0, the result of Ramalingam's approach is 0.299, since it cannot be recognized that node **6** is always entered via path **[3,4,6]**.

Table 2. Running example: Result of the history-independent approach.

$Prob(u, \delta)$	d_1	d_2	d_3	d_4	A
s	0	0	0	0	1
1	0.082	0.299	0.817	0.332	10
2	0.082	0.299	0.817	0.332	9
3	0.082	0.299	0.817	0.332	1
4	0.908	0.369	0.082	0.299	10
5	0.908	0.369	0.082	0.299	9
6	0.908	0.369	0.082	0.299	1
7	0.091	0.332	0.908	0.369	10
8	0.091	0.332	0.908	0.369	1

Table 3 lists the absolute deviations as a percentage. Except for start node **s** the data facts deviate in a range between $\pm 0.8\%$ and $\pm 91.9\%$. In the following section we present a novel approach which takes execution history into account and yields in this way significantly better results.

4 Two-Edge Approach

The main idea of our two-edge approach is to relate outgoing edges with incoming ones. Instead of propagating information through nodes the two-edge approach carries data flow information along edges in order to take execution history

Table 3. Comparison of the History-Independent Approach vs. Abstract Run (0% means that there is no deviation; maximum deviations are $\pm 100\%$).

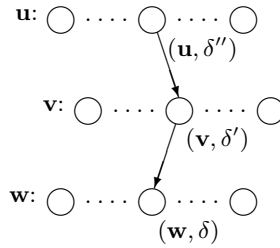
$\Delta\%$	d_1	d_2	d_3	d_4
s	0.0	0.0	0.0	0.0
1	8.2	29.9	-8.3	33.2
2	8.2	29.9	-7.2	33.2
3	8.2	29.9	-18.3	33.2
4	0.8	26.9	-1.8	29.9
5	-9.2	36.9	8.2	29.9
6	90.8	-63.1	-91.8	29.9
7	9.1	33.2	-9.2	26.9
8	9.1	33.2	-9.2	-63.1

into account. This is achieved by relating unknowns of the equation system to *EFCG* edges. Let $\hat{y}(\mathbf{v} \rightarrow \mathbf{w}, d)$ denote the expected frequency for fact d to hold true at node \mathbf{w} under the condition that edge $\mathbf{v} \rightarrow \mathbf{w}$ has been taken, and let $\hat{y}(\mathbf{v} \rightarrow \mathbf{w}, A)$ denote the expected number of times that edge $\mathbf{v} \rightarrow \mathbf{w}$ is executed.

Further, $p(u, v, w)$ denotes the probability that execution will follow edge $\mathbf{v} \rightarrow \mathbf{w}$ once it reaches edge $\mathbf{u} \rightarrow \mathbf{v}$. Consequently, the sum of the probabilities for all outgoing edges of edge $\mathbf{u} \rightarrow \mathbf{v}$ must be either one or zero¹. Path profiling techniques necessary to compute $\text{occurs}([\mathbf{u}, \mathbf{v}, \mathbf{w}], \pi_r)$ are discussed in detail in [19].

$$p(u, v, w) = \begin{cases} \frac{\text{occurs}([\mathbf{u}, \mathbf{v}, \mathbf{w}], \pi_r)}{\text{occurs}(\mathbf{u} \rightarrow \mathbf{v}, \pi_r)} & \text{if } \text{occurs}(\mathbf{u} \rightarrow \mathbf{v}, \pi_r) \neq 0, \\ 0 & \text{otherwise.} \end{cases} \quad (2)$$

Next we introduce a function **In** to determine the set of preceding ingoing edges of an edge. The function is derived from the exploded CFG. Fig. 3 depicts the relation between an edge $\mathbf{v} \rightarrow \mathbf{w}$ with data fact δ and its preceding node \mathbf{u} with data fact δ'' .

**Fig. 3.** Graphical visualization of function **In**

¹ If there are no outgoing edges for edge $\mathbf{u} \rightarrow \mathbf{v}$.

Hence, \mathbf{In} is given as follows,

$$\mathbf{In}(\mathbf{v} \rightarrow \mathbf{w}, \delta) = \{(u, \delta') | (u, \delta'') \rightarrow (v, \delta') \rightarrow (w, \delta) \in \Pi_{ECFG}\}. \quad (3)$$

where Π_{ECFG} represents the set of paths in the exploded control flow graph.

Since we need a “root” edge, we introduce an artificial edge $\epsilon \rightarrow \mathbf{s}$ with pseudo node ϵ . This artificial edge does not have any predecessor edge and the outgoing edges of node \mathbf{s} are successor edges. Note that $\text{pred}(\mathbf{s}) = \{\epsilon\}$. We associate the initial values $c(d)$ of the data flow analysis problem with the data facts of edge $\epsilon \rightarrow \mathbf{s}$. We further extend the probability $p(\epsilon, \mathbf{s}, v)$: For node \mathbf{v} , $p(\epsilon, \mathbf{s}, v)$ is either one (the program executes node v immediately after node \mathbf{s}) or the probability is zero (another subsequent node of node \mathbf{s} was taken.)

Finally, we can describe the linear equation system for a general data flow problem by the following formula. Once the equation system of edge unknowns has been solved, the expected frequencies of data facts are determined by summing up the unknowns of the incoming edges as shown in Equation 7

Two-Edge Equation System

$$\hat{y}(\epsilon \rightarrow \mathbf{s}, A) = 1 \quad (4)$$

for all d in D :

$$\hat{y}(\epsilon \rightarrow \mathbf{s}, d) = c(d) \quad (5)$$

for all $\mathbf{v} \rightarrow \mathbf{w}$ in E : for all δ in D_A :

$$\hat{y}(\mathbf{v} \rightarrow \mathbf{w}, \delta) = \sum_{(u, \delta') \in \mathbf{In}(\mathbf{v} \rightarrow \mathbf{w}, \delta)} p(u, v, w) * \hat{y}(\mathbf{u} \rightarrow \mathbf{v}, \delta') \quad (6)$$

for all w in N : for all δ in D_A :

$$\hat{y}(w, \delta) = \sum_{u \in \mathbf{pred}(v)} \hat{y}(\mathbf{v} \rightarrow \mathbf{w}, \delta) \quad (7)$$

The two-edge approach generates a set of very simple linear equations. Consequently, any of the standard algorithms for solving linear algebraic equations can be used. Most of these algorithms have a worst case complexity of $O(n^3)$ where n is the number of unknowns in the equation system. In the equation system of the two-edge approach there exist $(|E| + 1) \times |D_A|$ unknowns. Clearly, Ramalingam’s approach has less unknowns $|N| \times |D_A|$ due to the fact that the probabilities are related to nodes rather than edges. But the effort can be reduced by solving the equation system in two steps. In the first step we only solve A unknowns, which only depend on A unknowns itself. In the second step we solve data fact unknowns, which depend on A and data fact unknowns. Due to the first step A unknowns become constants for the second step.

Standard algorithms for solving linear equation system are usually too inefficient because they fail to utilize the extreme sparsity of the CFG. For our purpose, we can adapt various elimination methods [16, 117, 8] (a good survey

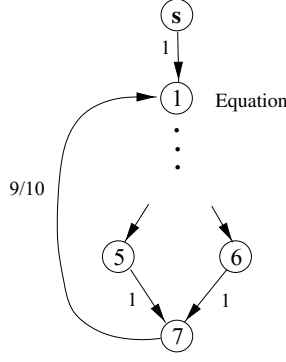


Fig. 4. Subgraph of Fig. 4 for node 1 annotated with edge probabilities.

can be found in [15]). These algorithms are often linear or almost linear in size of the graph.

Clearly, the two-edge approach can be extended to a three-edge, four-edge, or k-edge approach accordingly resulting in better probabilistic results. However, the time required to solve the system of equations will increase as well. As shown in our experimental section the two-edge approach yields for the SPECint95 benchmark suite very precise results. Hence, we believe that the two-edge approach is a good compromise between complexity and required precision.

In the following we present the differences between Ramalingam's and our two-edge approach for our running example. We illustrate the differences by discussing equations of data flow facts. In the first case Ramalingam's approach overestimates the probabilities and in the second case underestimates them.

Case 1. Consider Fig. 4 and the equation at node 1 for definition d_1 . Since the edge probability of $7 \rightarrow 1$ equals 9/10, the equations of Ramalingam's approach are given as follows:

$$\begin{aligned} y(1, d_1) &= 9/10 * y(7, d_1) + 1 * y(s, d_1) \\ y(7, d_1) &= y(5, d_1) + y(6, d_1) \end{aligned}$$

Since $y(s, d_1)$ is initialized to zero, $y(1, d_1)$ depends solely on $y(7, d_1)$. Further, definition d_1 is killed on edge $5 \rightarrow 7$ which results in $y(5, d_1)$ to be zero and, hence, $y(7, d_1)$ depends in turn solely on $y(6, d_1)$. Thus the value of $y(6, d_1)$ is propagated to $y(1, d_1)$, although the path $[6, 7, 1]$ is never executed. As a consequence, the value of $y(1, d_1)$ is too high compared with the result of the abstract run.

Since in the two-edge approach the unknowns are related to edges rather than nodes, the expected frequency of a data fact at a node is defined by the sum of unknowns of all incoming edges:

$$\hat{y}(1, d_1) = \hat{y}(s \rightarrow 1, d_1) + \hat{y}(7 \rightarrow 1, d_1)$$

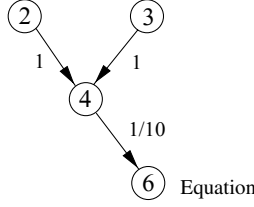


Fig. 5. Subgraph of Fig. 1 for node 6 annotated with edge probabilities.

Note that $\hat{y}(s \rightarrow 1, d_1)$ is 0 because there is no reaching definition in start node s . Only the second part of the sum needs to be considered.

$$\hat{y}(7 \rightarrow 1, d_1) = p(5, 7, 1) * \hat{y}(5 \rightarrow 7, d_1) + p(6, 7, 1) * \hat{y}(6 \rightarrow 7, d_1)$$

The first part of the sum reduces to 0 since definition d_1 is killed on edge $5 \rightarrow 7$. The second part of the sum is 0 as well due to the fact that probability $p(6, 7, 1)$ is zero (path $[6, 7, 1]$ is never taken). We obtain $\hat{y}(1, d_1) = 0$, which perfectly matches the result of the abstract run.

Case 2. Consider Fig. 5 and the data flow equation at node 6 for definition d_2 . Since the edge probability of $4 \rightarrow 6$ is given by $1/10$, we have for Ramalingam's approach the following equation:

$$y(6, d_2) = 1/10 * y(4, d_2)$$

Note that on edge $3 \rightarrow 4$ variable Y is defined by d_2 , whereas edge $2 \rightarrow 4$ is transparent for variable Y . Hence, definition d_2 can reach node 6 also via path $[2, 4, 6]$ depending on execution history. Since execution history is not taken into account, this results in a value less than probability 1 of the abstract run.

Again, for the two-edge approach, we are relating the unknowns to edges resulting in $\hat{y}(6, d_2) = \hat{y}(4 \rightarrow 6, d_2)$, since node 4 is the only predecessor. By weighting the incoming edges of node 4 with their probabilities we get:

$$\hat{y}(4 \rightarrow 6, d_2) = p(2, 4, 6) * \hat{y}(2 \rightarrow 4, d_2) + p(3, 4, 6) * \hat{y}(3 \rightarrow 4, d_2).$$

Note that probability $p(2, 4, 6)$ is zero and the first part of the sum vanishes since path $[2, 4, 6]$ is never taken. Probability $p(3, 4, 6)$ is 1 according to Equation 2 since path $[3, 4, 6]$ is always executed when edge $3 \rightarrow 4$ is reached. Therefore we obtain

$$\hat{y}(4 \rightarrow 6, d_2) = \hat{y}(3 \rightarrow 4, d_2)$$

Definition d_2 is executed on edge $3 \rightarrow 4$ and, hence, $\hat{y}(3 \rightarrow 4, d_2)$ is equal to $\hat{y}(3 \rightarrow 4, \Lambda)$. Since $\hat{y}(3 \rightarrow 4, \Lambda)$ denotes the number of times edge $3 \rightarrow 4$ occurs

in the program run, $\hat{y}(\mathbf{3} \rightarrow \mathbf{4}, \mathbf{1})$ equals 1. Finally, we substitute backwards and gain $\hat{y}(\mathbf{6}, d_2) = 1$ which perfectly matches the result of the abstract run.

It is important to stress that our method, which is based on relating outgoing edges to incoming ones, can trace rather complicated flow graph paths. E.g. we get that definition d_3 reaches node **6** each time (i.e. probability equals 1): Thus our method finds out that edge $\mathbf{5} \rightarrow \mathbf{7}$ has been executed prior to node **6** in one of the previous loop iterations and that definition d_3 has not been killed by taking edge $\mathbf{2} \rightarrow \mathbf{4}$ before execution arrives at node **6**.

For the running example the solution of our two-edge approach is identical to the best solution \mathcal{S}_{best} of the abstract run as shown in Table [11](#) whereas Ramalingam’s approach significantly deviates from the abstract run as shown in Table [13](#).

5 Experimental Results

In our experiments we address two issues. First, we show that for the two-edge approach the analysis results are significantly better. We illustrate this by analyzing the SPECint95 benchmark suite. Second, we demonstrate that probabilistic data flow analysis frameworks are viable even for larger programs, since in most cases the original equation system can be reduced considerably and the remaining equations are usually trivial ones with a constant or only one variable on the right-hand side.

The compilation platform for our experiments is GNU gcc. We integrated abstract run, Ramalingam’s one-edge approach, and the two-edge approach. To evaluate Ramalingam’s approach and the two-edge approach we have chosen SPECint95 as benchmark suite and the reaching definitions problem as reference data flow problem. The profile information has been generated by running the training set of SPECint95. Of course, the same training set has been used for the abstract run as well.

Probabilistic Data Flow Results. In our first experiment we have compared the deviations of the two-edge approach from the abstract run with the deviations of the one-edge approach from the abstract run. For each benchmark program of SPECint95 we added the absolute deviations for each CFG node and each data flow fact. The ratio of the sums of the two-edge approach over the one-edge approach is shown in Fig. [6](#). The improvement for SPECint95 is in the range of 1.38 for *vortex* up to 9.55 for *li*. The experiment shows that the results of the two-edge approach compared to the one-edge approach are significantly better.

Above we have shown that the results of the two-edge approach are significantly better than the results of the one-edge approach, but maybe both solutions deviate substantially from the theoretically best solution. Hence, we executed the abstract run to get the theoretically best solution and compared it with the two-edge approach. For each benchmark program of SPECint95 we calculated for all CFG nodes and data flow facts the mean of deviations of the probabilities. The results are shown in Fig. [7](#). The mean of deviations is between

SPECint95	Improvement Ratio
go	1.95
m88ksim	2.26
gcc	2.54
compress	3.28
li	9.55
jpeg	4.20
perl	2.44
vortex	1.38

Fig. 6. SPECint95: Improvement ratio of two-edge approach compared to Ramalingam’s one-edge approach.

SPECint95	Av. Prob Δ	Function hits
go	1.72	42.6%
m88ksim	0.31	87.9%
gcc	0.41	76.8%
compress	0.36	78.9%
li	0.11	94.2%
jpeg	0.20	92.2%
perl	0.16	89.9%
vortex	0.16	92.4%

Fig. 7. SPECint95: Comparison two-edge approach with abstract run.

0.11% for *li* and 1.72% for *go*. The reason for this excellent result is that usually programs execute only a small fraction of the potential paths. For non-executed CFG nodes the result of the two-edge approach always coincides with the probability of the abstract run (namely zero), since the two-edge probability (Equ. 2) yields zero. Hence, we did a second comparison which is more meaningful. We calculated the percentage of functions for which the two-edge approach coincides with the abstract run. The function hits reaches for the two-edge approach from 42.6% for benchmark *go* up to 94.2% for *li* which is an excellent result as well.

Effort for Solving Linear Algebraic Equation Systems. In general, the worst case complexity for solving linear algebraic equation systems is $O(n^3)$ with n denoting the number of unknowns. The number of unknowns for the original one-edge and two-edge equation system as described in Sections 3 and 4 can be rather big. However, unknowns, which are related to control flow nodes or edges which are not visited during the profile run, can be removed immediately. In the SPECint95 suite only 48.2% of the original unknowns of the one-edge approach have to be computed; for the two-edge approach 40.5% of the original unknowns need to be solved.

Moreover, the structure of the equations for the SPECint95 suite is rather simple. For the one-edge approach 54.3% of the equations are trivial ones with

constants on the right-hand side only. Similarly, for the two-edge approach we have a percentage rate of 58.8% equations with constants on the right-hand side. For both approaches only about 1.6% of the equations have more than two variables (up to 150) on the right-hand side. Hence, a linear equation solver for sparse systems is of key importance.

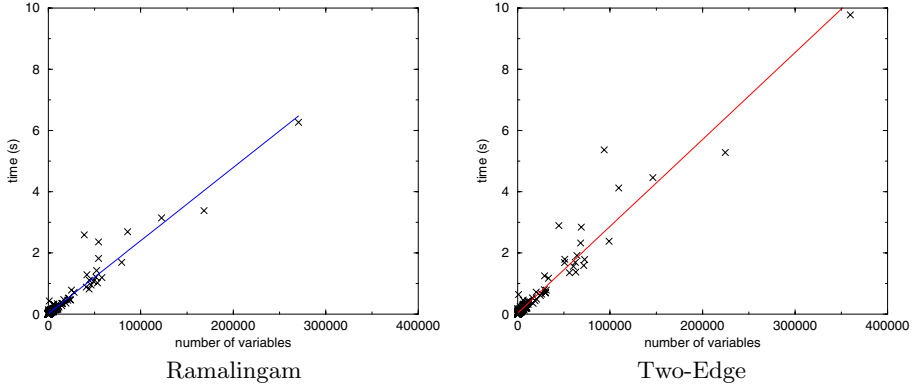


Fig. 8. Time to solve unknowns.

We have chosen an elimination framework introduced by Sreedhar et al. [16], which was originally developed to solve classical bit-vector problems on control flow graphs. Especially, for our extremely sparse equation system Sreedhar’s framework is well suited. In Fig. 8 a data point represents the number of unknowns for a C-module of SPECint95 and the time in seconds to solve the unknowns. Here, we have measured the graph reduction and propagation of Sreedhar’s framework without setting up DJ-graphs² and without setting up the equations itself which takes additionally time since profile data must be accessed. The left graph depicts the measurements of Ramalingam’s approach – the right graph shows the measurements of the two-edge approach. It is really remarkable that Sreedhar’s algorithm nearly works linear on extremely sparse equation systems. The measurements were taken on a Sun Ultra Enterprise 450 (4x UltraSPARC-II 296MHz) with 2560MB RAM.

6 Related Work

Several approaches have been proposed which take advantage of profile information to produce highly efficient code.

Ramalingam [13] presents a generic data flow framework which computes the probability that a data flow fact will hold at some program point for finite

² To set up DJ-graphs dominator trees are required. Recently, linear algorithms were introduced [4].

bi-distributive subset problems. The framework is based on exploded control flow graphs introduced by Reps, Horwitz, Sagiv [14] and on Markov-chains. Contrary to our approach, execution history is not taken into account. To our best knowledge we are not aware of any other execution history based approach. Optimizations based on PDFA's are presented in [10,11].

Alternatively, Ammons and Larus [2] describe an approach to improve the results of data flow analysis by identifying and duplicating hot paths in the program's control flow graph resulting in a so-called hot path graph in which these paths are isolated. Data flow analysis applied to a hot path graph yields more precise data flow information. The goal of this approach differs from our work. We improve the precision of a probabilistic data flow solution and do not modify the control flow graph in order to enable heavily executed code to be highly optimized.

Finally, profile information is used by several researchers for specific optimization problems in order to get better results (e.g. [6,5,12,7,3,18]).

7 Conclusion

Probabilistic data flow frameworks set forth new directions in the field of optimization. We presented a novel, practicable probabilistic data flow framework which takes execution history into account by relating outgoing edges to incoming ones. In this way we achieve significantly better results. Practical experiments which have been performed for the SPECint95 benchmark suite showed that the two-edge approach is feasible and the precision of the probabilistic results is sufficient.

References

1. F. E. Allen and J. Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137–147, March 1976.
2. G. Ammons and J.R. Larus. Improving data-flow analysis with path profiles. In *Proc. of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI'98)*, pages 72–84, Montreal, Canada, June 1998.
3. R. Bodík, R. Gupta, and M.L. Soffa. Complete removal of redundant expressions. *ACM SIGPLAN Notices*, 33(5):1–14, May 1998.
4. A.L. Buchsbaum, H. Kaplan, A. Rogers, and J.R. Westbrook. Linear-time pointer-machine algorithms for LCAs, MST verification, and dominators. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing (STOC-98)*, pages 279–288, New York, May 23–26 1998. ACM Press.
5. B. Calder and D. Grunwald. Reducing branch costs via branch alignment. *ACM SIGPLAN Notices*, 29(11):242–251, November 1994.
6. J. A. Fisher. Trace scheduling : A technique for global microcode compaction. *IEEE Trans. Comput.*, C-30(7):478–490, 1981.
7. R. Gupta, D. Berson, and J.Z. Fang. Path profile guided partial dead code elimination using predication. In *International Conference on Parallel Architectures and Compilation Techniques (PACT'97)*, pages 102–115, San Francisco, California, November 1997.

8. M.S. Hecht. *Flow Analysis of Computer Programs*. Programming Language Series. North-Holland, 1977.
9. E. Mehofer and B. Scholz. Probabilistic data flow system with two-edge profiling. *Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo'00)*. *ACM SIGPLAN Notices*, 35(7):65 – 72, July 2000.
10. E. Mehofer and B. Scholz. Probabilistic procedure cloning for high-performance systems. In *12th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'2000)*, Sao Pedro, Brazil, October 2000.
11. E. Mehofer and B. Scholz. Probabilistic communication optimizations and parallelization for distributed-memory systems. In *PDP 2001*, Mantova, Italy, February 2001.
12. T. C. Mowry and C.-K. Luk. Predicting data cache misses in non-numeric applications through correlation profiling. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-97)*, pages 314–320, Los Alamitos, December 1–3 1997. IEEE Computer Society.
13. G. Ramalingam. Data flow frequency analysis. In *Proc. of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI'96)*, pages 267–277, Philadelphia, Pennsylvania, May 1996.
14. T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proc. of the ACM Symposium on Principles of Programming Languages (POPL'95)*, pages 49–61, San Francisco, CA, January 1995.
15. B. G. Ryder and M. C. Paull. Elimination algorithms for data flow analysis. *ACM Computing Surveys*, 18(3):277–315, September 1986.
16. V. C. Sreedhar, G. R. Gao, and Y.-F. Lee. A new framework for elimination-based data flow analysis using DJ graphs. *ACM Transactions on Programming Languages and Systems*, 20(2):388–435, March 1998.
17. R. E. Tarjan. Fast algorithms for solving path problems. *Journal of the ACM*, 28(3):594–614, July 1981.
18. C. Young and M. D. Smith. Better global scheduling using path profiles. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-98)*, pages 115–126, Los Alamitos, November 30–December 2 1998. IEEE Computer Society.
19. C. Young and M.D. Smith. Static correlated branch prediction. *ACM Transactions on Programming Languages and Systems*, 21(5):1028–1075, September 1999.

Imperative Program Transformation by Rewriting

David Lacey and Oege de Moor

Oxford University Computing Laboratory

Abstract. We present a method of specifying standard imperative program optimisations as a rewrite system. To achieve this we have extended the idea of matching sub-terms in expressions with simple patterns to matching blocks in a control flow graph. In order to express the complex restrictions on the applicability of these rewrites we add temporal logic side conditions. The combination of these features allows a flexible, high level, yet executable specification of many of the transformations found in optimising compilers.

1 Introduction

Traditional rewrite systems are good for expressing transformations because:

- Rewrites tend to be very succinct in their specification.
- The represented transformation is intuitively expressed. Rewrite patterns express the required partial structure of transformable terms in an explicit way.
- There are existing approaches to combining rewrite sets via strategies and reasoning about their properties such as confluence, soundness and termination.

A lot of the success of expressing program transformations by rewriting has been for purely functional programs. However, rewriting imperative programs is problematic. One of the main reasons for this is that the conditions under which it is possible to apply a rewrite are hard to describe. Many rewrite systems work by transforming abstract syntax trees and to decide applicability one needs complex definitions to analyse these trees. The conditions are easier to specify if the program is represented by its control flow graph but it is not obvious how to specify rewrites on graphs.

This paper tackles these problems in two ways. Firstly, a pattern matching language is developed that detects patterns in programs represented by their control flow graph. Secondly, we use a language for specifying applicability conditions on rewrites using temporal logic constructs for reasoning over the graph. This leads to a language which is capable of expressing many common transformations on imperative programs found in optimising compilers. To give an impression of our approach, here is the specification (which will be described in detail during the paper) of the optimising transformation *constant propagation*.

It says that an assignment $x := v$ (where v is a variable) can be replaced by $x := c$ if the “last assignment” to v was $v := c$ (where c is a constant). The side condition formalises the notion of “last assignment”, and will be explained later in the paper:

$$\begin{array}{l} n : (x := v) \implies x := c \\ \text{if} \\ n \vdash A^\Delta(\neg def(v) \, U \, def(v) \wedge stmt(v := c)) \\ \text{conlit}(c) \end{array}$$

The rewrite language has several important properties:

- The specification is in the form of a rewrite system with the advantages of succinctness and intuitiveness mentioned above.
- The rewrite system works over a control flow graph representation of the program. It does this by identifying and manipulating *graph blocks* which are based on the idea of basic blocks but with finer granularity.
- The rewrites are executable. An implementation exists to automatically determine when the rewrite applies and to perform the transformation just from the specification.
- The relation between the conditions on the control flow graph and the operational semantics of the program seems to lend itself to formal reasoning about the transformation.

The paper is organised as follows. §2 covers earlier work in the area and provides the motivation for this work. §3 describes our method of rewriting over control graphs. §4 describes the form of side conditions for those rewrites. §5 gives three examples of common transformations and their application when given as rewrites. §6 discusses what has been achieved and possible applications of this work.

2 Background

Implementing optimising transformations is hard: building a good optimising compiler is a major effort. If a programmer wishes to adapt a compiler to a particular task, for example to improve the optimisation of certain library calls, intricate knowledge of the compiler internals is necessary. This contrasts with the description of such optimisations in textbooks [13, 26], where they are often described in a few lines of informal English. It is not surprising, therefore, that the program transformation community has sought declarative ways of programming transformations, to enable experimentation without excessive implementation effort. The idea to describe program transformations by rewriting is almost as old as the subject itself. One early implementation can be found in the TAMPR system by Boyle, which has been under development since the early '70s [8, 9]. TAMPR starts with a specification, which is translated to pure lambda calculus, and rewriting is performed on the pure lambda expressions. Because programs

are represented in a functional notation, there is no need for complex side conditions, and the transformations are all of a local nature. OPTRAN is also based on rewriting, but it offers far more sophisticated pattern matching facilities [24]. A yet more modern system in the same tradition is Stratego, built by Visser [32]. Stratego has sophisticated mechanisms for building transformers from a set of labeled, unconditional rewrite rules. Again the published applications of Stratego are mostly restricted to the transformation of functional programs. TrafoLa is another system able to specify sophisticated syntactics program patterns [20].

It would be wrong, however, to suggest that rewriting cannot be applied in an imperative setting. For instance, the APTS system of Paige [27] describes program transformations as rewrite rules, with side conditions expressed as boolean functions on the abstract syntax tree, and data obtained by program analyses. These analyses also have to be coded by hand. This is the norm in similar systems that have been constructed in the high-performance computing community, such as MT1 [7], which is a tool for restructuring Fortran programs. Other transformation systems that suffer the same drawback include Khepera [15] and Txl [11].

It is well known from the compiler literature that the program analyses necessary in the side conditions of transformations are best expressed in terms of the control flow graph of an imperative program. This has led a number of researchers, in particular Assman [4] and Whitfield and Soffa [33] to investigate graph transformation as a basis for optimising imperative programs. Whitfield and Soffa's system, which is called Genesis, allows the specification of transformations in a language named Gospel. Gospel has very neat declarative specifications of side conditions (referring to the flow graph), but modifications of the program are carried out in a procedural manner. Assmann's work, by contrast, is much more declarative in nature since it relies on a quite general notion of graph rewriting. One advantage of this approach is that certain conditions on the context can be encoded as syntactic patterns. It is impossible, however, to make assertions about program paths. This restriction is shared by Datalog-like systems expressing program analyses as logic programs [12].

A number of researchers have almost exclusively concentrated on elegant ways of expressing the side conditions of transformations, without specifying how the accompanying transformations are carried out. For example, Sharlit [30] is a tool for generating efficient data flow analyses in C. It is much more expressive than the limited notation of the tools discussed above, but the user has to supply flow functions in C, so the specifications are far from declarative. However, several systems (for example BANE [2] and PAG [25]) provide a more elegant system of automatically generating dataflow information from recursive sets of dataflow equations.

The work of Bernhard Steffen provides a new approach in the field. In a pioneering paper [28], Steffen showed how data flow analyses could be specified through formulae in temporal logic. Steffen's descriptions are extremely concise and intuitive. In a series of later papers, Steffen and his coworkers have further articulated the theory, and demonstrated its practical benefits [22,23,29].

The present paper builds on Steffen's ideas, combining it with a simple notion of rewriting on control flow graphs, and introducing a kind of logical variable so that the applicability conditions can be used to instantiate variables on the right-hand side of a rewrite rule.

3 Rewriting Flow Graphs

3.1 Rewrite Systems

Rewrites specify a transformation between two objects. Usually these objects are tree-like expressions but they may also be more general graph structures. They consist of a left hand side pattern, a right hand side pattern and (optionally) a condition. We shall express this in the following manner:

$$LHS \Longrightarrow RHS \quad \text{if} \quad Condition$$

A pattern expresses a partial structure of an object. It will contain free variables, denoting parts of the structure that are unknown. For each pattern there will be a set of objects that have the same partial structure as the pattern. Objects of this set *match* the pattern. Any rewrite system needs a matching algorithm that determines whether an object matches a pattern. When a pattern matches then the free variables in the pattern will correspond to known parts of the matching object. So we can talk about the *matching substitution* that maps the free variables to these known parts.

The rewrite is implemented by finding a sub-object of the object we are transforming which matches the left hand side pattern with matching substitution θ . The matching sub-object is replaced by the right hand side, where variables are instantiated by θ .

3.2 Representations of Programs

The imperative programs we wish to transform consist of program statements linked together by various control structures. To transform these programs we have a choice of representations. The program statements themselves are naturally expressed as syntax trees but this is not necessarily the best way to represent the control structure of a program. Figure 1 gives an example of a simple program and two possible representations.

Expression Trees. Expression trees are the most common objects used for rewriting. The advantage is that sub-term replacement is a simple operation. However, reasoning about dataflow properties is difficult in this representation.

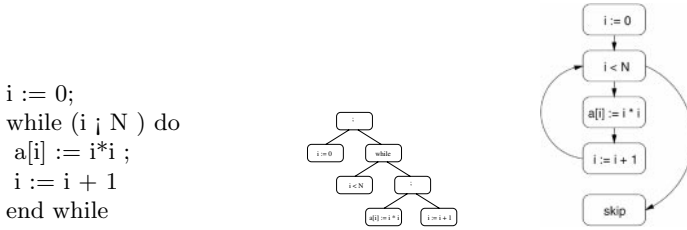


Fig. 1. A simple program and two representations

Control Flow Graphs. The control flow graph treats statements as nodes and edges as possible control flow of the program. If the edges are labelled for conditional jumps it can characterise the program. This allows side conditions to be expressed more simply but patterns over graphs are more complicated. For many structured transformations these pattern matching problems can be overcome however, so this is the representation we chose to work with.

Often the control flow graph is used as an intermediate representation in compilers on which optimising transformations are performed. This is another reason for choosing the CFG as our representation.

The representation chosen here is over a fairly simple and naive language. In particular there is no notion of pointers or aliasing.

3.3 Graph Blocks and Graph Rewriting

We concentrate on the notion of single-entry-single-exit regions as our objects of rewriting. For convenience we will refer to these regions as *graph blocks*. The term *hammocks* is sometimes used for this, but the use of this terminology is inconsistent, as pointed out in [21]. Often analysis in compilers is in terms of *basic blocks*, these are also graph blocks but tend to be specified as not containing cycles and maximal with respect to this property. We do not put this restriction on graph blocks since we need a finer granularity to specify transformations.

Any single node is a graph block, as is an entire program. Also, for many structured programs (not containing arbitrary goto statements), any sequence of statements also corresponds to a block in the control flow graph.

The restriction of objects for rewriting from general graphs to graph blocks allows us to easily specify a language for expressing patterns over graph blocks. The most elementary pattern is just a free variable representing any graph block and we will represent these variables by lower case Roman letters.

Any single node is a graph block, so one would like a way of specifying a single node. The important property we wish to match for a node is its associated program statement. Since this is just a term we can create patterns for these in the usual way. We can then specify in block patterns that a statement pattern matches any graph block consisting of a single node that matches the statement

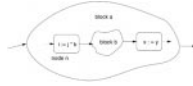


Fig. 2. A graph block pattern

pattern¹. Sometimes, it is useful to have a free variable which gives a name to such a block, in which case the statement pattern is prefixed by a free variable and a colon. Here are a couple of examples:

$$\begin{aligned} x &:= i \\ n : (y &:= z * x) \end{aligned}$$

The simplest way in which blocks are related is by sequencing together. This is expressed by the `;` operator in the pattern language. So a pattern $a; b$ will match the union of a block which matches a and a block which matches b , providing the exit of a has a sequential successor that is the entry to b .

The final operator we used in our matching language is that of a *context*. This is used to express the idea of a block that contains another block. So the pattern $a[b]$ will match a block that matches a , and furthermore contains a sub-block matching b .

Overall, the grammar of the matching language is:

$$\begin{aligned} \langle \text{block pattern} \rangle &::= \langle \text{var} \rangle \\ &\quad | \langle \text{var} \rangle : \langle \text{statement pattern} \rangle \\ &\quad | \langle \text{var} \rangle [\langle \text{block pattern} \rangle] \\ &\quad | \langle \text{block pattern} \rangle ; \langle \text{block pattern} \rangle \end{aligned}$$

To illustrate, Figure 2 pictorially shows the pattern:

$$a[n : (i := j * k); b; x := y]$$

During rewriting a pattern may match with an empty subgraph. In this case we see this as being equivalent to matching a single node which performs no operation. This is required for construction of the new graph created by the rewrite.

4 Side Conditions

The applicability conditions of a rewrite are expressed as side conditions which place restrictions on what objects can be matched to the free variables in the left hand side pattern of the rewrite. The language of conditions presented here is not claimed in any way to be complete, but it suffices to specify many of the standard transformations found in optimising compilers.

¹ There is an ambiguity here as a statement pattern could also be just a single free variable, but we stipulate that this is a block pattern and not a statement pattern.

These restrictions are specified by propositions containing the free variables found in the rewrite which must hold of the objects matching those variables. There are simple restrictions one may make about expressions used by statements such as whether they consist of just a constant or variable literal. These primitive conditions are listed in Figure 3. For example, one may specify the following rewrite which performs a limited form of constant propagation:

$$x := c; y := x \Longrightarrow x := c; y := c \text{ if } \text{conlit}(c)$$

These basic conditions can be combined with standard propositional logic operators (\wedge, \vee, \neg) to form more complex propositions. For example, here is the same limited constant propagation rewrite that only propagates something that is a constant literal and has type *ShortInt*:

$$x := c; y := x \Longrightarrow x := c; y := c \text{ if } \text{conlit}(c) \wedge \text{type}(c, \text{ShortInt})$$

<i>True</i>	Always holds
<i>False</i>	Never holds
<i>conlit</i> (<i>x</i>)	Holds if <i>x</i> is a constant literal
<i>varlit</i> (<i>x</i>)	Holds if <i>x</i> is a variable literal
<i>type</i> (<i>x</i> , <i>t</i>)	Holds if <i>x</i> is of type <i>t</i>

Fig. 3. Basic conditions

These conditions allow us to specify restrictions on statements in the control flow graph but not on how the nodes in the graph relate to each other. The paper by Steffen [29] shows that temporal logic is a very succinct way of specifying dataflow properties. This is the approach we take here.

The restriction on nodes in the control flow graph are expressed as a sequent. This is a formula of the form:

$$n \vdash \text{TempForm}$$

Where *TempForm* is a temporal formula whose syntax is described below. A sequent can be read as saying that a formula “holds at” or “is satisfied by” a particular node.

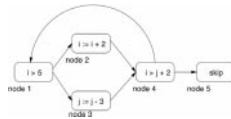


Fig. 4. A sample control flow graph

$def(x)$	The variable x is defined at this node.
$use(x)$	The variable x is used at this node.
$node(n)$	This node is n
$stmt(p)$	The statement associated with this node matches pattern p

Fig. 5. Basic temporal conditions

The most basic things we want to know about a node are its identity and the associated program statement. The syntax for the latter is:

$$n \vdash stmt(StmtPattern)$$

So the following formula would hold in Figure 4 with the substitution $\{n \mapsto node\ 2, x \mapsto i, a \mapsto i, b \mapsto 2\}$:

$$n \vdash stmt(x := a + b)$$

A couple of useful predicates derived from $stmt$ are $def(x)$ and $use(x)$, these are described in Figure 5

Identity is specified using the temporal predicate $node$. For example:

$$n \vdash node(m)$$

This formula holds for any substitution that maps n and m to the same node. This facility is useful when combined with the other temporal constructors.

We need ways to relate nodes to each other. This is done via the four temporal constructors described in Figure 6. The first and third, EX and AX specify the relation between a node and its immediate successors. For example, the formula below states that node n has a successor that is node m , thus restricting n and m to be related as such in the control flow graph:

$$n \vdash EX(node(m))$$

In Figure 4, a substitution satisfying this formula is $\{m \mapsto node\ 4, n \mapsto node\ 3\}$.

We may not be interested in only successors to a node but predecessors also. Putting a Δ next to a constructor specifies predecessors instead of successors. For example, the same relation as above can be expressed as:

$$m \vdash EX^\Delta(node(n))$$

These constructors express relations between *immediate* successors or predecessors. However, many relations are between nodes which have paths of multiple edges between them. To express these relations we appeal to the *until* operators of computational tree logic ($A(\dots U \dots), E(\dots U \dots)$) [10]. These are predicates on paths in the control flow graph. A path is a (possibly infinite) sequence of

$EX(f_1)$	There exists a successor of this node such that f_1 is satisfied at that successor.
$E(f_1 U f_2)$	There exists a path from this node to a node n such that every node on that path up to but not including n satisfies f_1 and n satisfies f_2 .
$AX(f_1)$	All successors of this node satisfy f_1
$A(f_1 U f_2)$	Every path from this node is either infinite with all the nodes on the path satisfying f_1 , or finite such that f_1 is satisfied on every node until a node that satisfies f_2 .

Fig. 6. Temporal constructors

nodes $< n_1, n_2, \dots >$ such that each consecutive pair (n_i, n_{i+1}) is an edge in the control flow graph. The formula $n \vdash E(f_1 U f_2)$ holds if a path exists, starting at n such that f_1 is true on this path until f_2 is true on the final node. That is, we have a finite non-empty path $< n_1, n_2, \dots, n_N >$ such that $n_1 = n$, for all $1 \leq i \leq N - 1$: $n_i \vdash f_1$ and $n_N \vdash f_2$. For example the following formula holds if there is a path from node n to node m such that every node on that path does not define x ²:

$$n \vdash E(\neg def(x) U node(m))$$

For example, this formula would match the graph in Figure 4 with substitution $\{n \mapsto node\ 3, x \mapsto i, m \mapsto node\ 1\}$ ³.

The universal until operator: $n \vdash A(f_1 U f_2)$, says that every path starting at n is either infinite with every node satisfying f_1 or finite with a prefix that satisfies f_1 until a point where f_2 is satisfied. Note that here the logic deviates slightly for standard CTL in that in the universal case we use the weak version of the until operator.

As with the EX/AX constructors we can look at paths that follow predecessor links instead of successor links. So $E^\Delta(\dots U \dots)$ and $A^\Delta(\dots U \dots)$ look at paths running backwards through the control flow graph.

The temporal operators find paths over the entire control flow graph. Sometimes it is necessary to restrict their scope. For example, the following formula says that all paths, *whose nodes all lie within the graph block b* , satisfy *True* until $def(x)$:

$$A[b](True U def(x))$$

This would hold in Figure 4 for substitution $\{a \mapsto nodes\ 1\ to\ 4, x \mapsto i\}$.

Sequents of the form $n \vdash TempForm$ are good for reasoning about nodes in the graphs. However, the block patterns described in section 3 can result in free

² Note here that the U operator has weakest precedence amongst the logical operators.

³ This is not the only substitution that will provide a match for that formula on that graph.

variables standing for graph blocks. We extend the idea of sequents to deal with graph blocks in the following four ways:

$$\begin{aligned} all(a) &\vdash TempForm \\ exists(a) &\vdash TempForm \\ entry(a) &\vdash TempForm \\ exit(a) &\vdash TempForm \end{aligned}$$

Respectively, these correspond to the statements “every node in the block satisfies ...”, “there exists a node in the block satisfying ...”, “the entry of the block satisfies ...” and “the exit of the block satisfies ...”.

Fresh Variables. In many rewrite systems the only free variables that can occur in a rewrite are on the left hand side of the rewrite. However, the free variables in a condition may be so constrained that they can be used to construct the right hand side without appearing in the left hand side. For example, the following rewrite has a side condition that all predecessors are of the form $v := c$. This will restrict the value of c so it can be used in the right hand side:

$$x := v \implies x := c \quad \text{if } AX^\Delta(stmt(v := c))$$

Our use of free variables in logical predicates (where there may be several possible satisfying substitutions) is similar to logic programming. It is a very important extension to the rewrite language, allowing conditions to interact more directly with the rewrite.

In line with the similarity to logic programming it is useful to have some evaluating predicates in the side conditions. In particular, the condition below states that x must equal the value of y multiplied by z :

$$x \text{ is } y \times z$$

This predicate is only meaningful when x, y and z match constant numeric literals.

5 Examples

5.1 Constant Propagation

Constant propagation is a transformation where the use of a variable can be replaced with the use of a constant known before the program is run (i.e. at compile time).

The standard method of finding out if the use of a variable is equivalent to the use of constant is to find all the possible statements where the variable could have been defined, and check that in all of these statements, the variable is assigned the same constant.

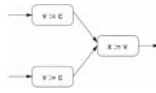


Fig. 7. Possibly transformable code snippet

The rewrite itself is simple⁴:

$$n : (x := v) \Longrightarrow x := c$$

Evidently we need restrictions on v and c . The idea is that if we follow all paths of computation backwards from node n , then the first definition of v we come to must be of the form $v := c$. The paths of computation backwards from node n are just the backwards paths from n in the control flow graph. We want to check all of them so the $A^\Delta(..U..)$ constructor is appropriate. To fit into the “until” path structure we can note that requiring the first definition on a path to fulfill a property is the same as saying the path satisfies non-definition until a point where it is at a definition and the property holds. This gives us the condition:

$$n : (x := v) \Longrightarrow x := c \text{ if } n \vdash A^\Delta(\neg def(v) U def(v) \wedge stmt(v := c)) \\ conlit(c)$$

The language we use to specify rewrites is quite powerful and, unsurprisingly, one can specify transforms in different ways which have subtle variations in behaviour. Another way of looking at constant propagations is that it rewrites a program consisting of a constant assignment to a variable which is itself used after some intermediate block of code. This leads us to form a rewrite thus:

$$v := c; a; x := v \Longrightarrow v := c; a; x := c$$

Here the condition needed is that v is not redefined in block a . This is simple to express as:

$$v := c; a; x := v \Longrightarrow v := c; a; x := c \text{ if } all(a) \vdash \neg def(v) \\ conlit(c)$$

However, this formulation will not transform the program fragment shown in Figure 7 (in that the graph does not match the LHS of the rewrite) whereas the former formulation would. This shows that different specifications can vary in quite subtle ways and reasoning about equivalence probably requires a more formal treatment.

⁴ This rewrite tackles one kind of use of the variable v . Other similar rewrites can be formed for other uses (e.g. as operands in a complex assignment)

5.2 Dead Code Elimination

Dead code elimination removes the definition of a variable if it is not going to be used in the future. The rewrite simply removes the definition:

$$n : (x := e) \Longrightarrow \text{skip}$$

The condition on this rewrite is that all future paths of computation do not use x . Another way of looking at this property is to say that there does not exist a path that can find a node that uses x . This can be specified using the $E(\dots U \dots)$ construct. However, care is required, since we do not care if x is used at node n . So we can specify that for all successors of n , there is no path of computation that uses x :

$$n : (x := e) \Longrightarrow \text{skip if } n \vdash AX(\neg E(\text{True } U \text{ use}(x)))$$

Note that for this rule and the one above, while providing illustrative examples, may not perform all the constant propagations or dead code elimination one may want. In particular, if a variable is not truly dead but *faint* [16]. These optimisations would require more complicated rules combined with other transformations such as copy propagation.

5.3 Strength Reduction

Strength reduction is a transformation that replaces multiplications within a loop structure into additions that compute the same value. This will be beneficial if the computational cost of multiplication is greater than that of addition.

For example, the following code:

```
i := 0;
while (i < N) do
  j := i * 3;
  a[i] := j;
  i := i + 1;
end while
```

Could be transformed to:

```
i := 0;
j := 0;
while (i < N) do
  a[i] := j;
  i := i + 1;
  j := j + 3;
end while
```

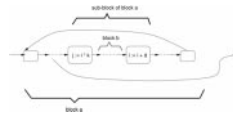


Fig. 8. Pattern for loop strengthening



Fig. 9. After the strengthening transformation

The pattern here is that a variable i is incremented by a constant each time in the loop and the variable j is calculated by multiplying i by a constant value. The pattern as it would occur in a control flow graph is shown in Figure 8. It can be captured using our graph matching language by:

$$a[n : (j := i * k); b; m : (i := i + d)]$$

The transformed code will be of the form given in Figure 9. This can be specified as:

$$j := i * k; a[b; i := i + d; j := j + step]$$

Obviously appropriate conditions are needed that restrict the value of $step$. This can be calculated as such:

$$step \text{ is } k \times d$$

For the rewrite to be valid we stipulate both k and d to be constant literals. Therefore we add the condition:

$$conlit(k) \wedge conlit(d)$$

The transformation depends on the fact that the node m and node n are the only points in the loop that define i or j respectively. This can be stipulated by saying that for every node in graph block a either we do not define i , or we are at node m and either we do not define j , or we are at node n :

$$all(a) \vdash \neg def(i) \vee node(m) \quad \wedge \quad all(a) \vdash \neg def(j) \vee node(n)$$

Finally, this rewrite can only produce a benefit if the block a is indeed a loop. To specify this we can use the observation that a is a loop if the entry to a has a predecessor in a . That property is conveniently phrased using a temporal operator that has reduced scope. If the entry to a has a predecessor within a then this predecessor must also be a descendant of the entry of a :

$$entry(a) \vdash EX^\Delta[a](True)$$

The condition is equivalent to saying that the entry to the block a is a loop header which has a back edge connected to it. Putting everything together, strength reduction is captured by:

$$\begin{aligned} & a[n : (j := i * k); b; m : (i := i + d)] \\ \implies & j := i * k; a[b; i := i + d; j := j + step] \\ \text{if} & \\ & \text{step is } k \times d \\ & \text{conlit}(k) \\ & \text{conlit}(d) \\ & all(a) \vdash \neg def(i) \vee node(m) \\ & all(a) \vdash \neg def(j) \vee node(n) \\ & entry(a) \vdash EX^\Delta[a](True) \end{aligned}$$

6 Discussion and Future Work

6.1 Summary

This paper has demonstrated with a few examples a method of specifying imperative program transformations as rewrites. It is the authors' belief that these specifications are clear and intuitive. The approach combines techniques from program optimisation, rewriting, logic programming and model checking.

6.2 Implementation

The rewrite language presented has been specifically designed to express executable specifications. A small prototype system has already been implemented covering some of the specification language to automatically execute rewrites on simple programs.

The implementation is built around a constraint solving system. A left hand pattern can be seen as a constraint on the free variables in the pattern. The side conditions can be viewed in the same way. The rewrite engine translates the left hand side pattern and the side conditions into one constraint set with is then resolved into normal form. The normal form for this constraint set provides a list of matching substitutions that obey the restrictions in the side conditions. An interesting aspect of the implementation is the path finding algorithm which implements model checking fixed point algorithms [10] raised to the level of constraints.

Currently, the implementation is not as efficient as if one had hand-coded the transformations. However, this is to be expected due to the general nature of our approach. Initial experimentation shows that the performance does not terminally degrade for non-trivial (~400 lines) sized programs. We believe that a workable efficiency could be achieved with more efficient constraint representations and the addition of code that caches and incrementally updates information

obtained while checking/applying different rewrites to the code. In addition it may be useful in future to extend to language to let the user specify which common side conditions could be factored out of the rewrites to be calculated together. Greater detail of the implementation should appear in a future paper.

6.3 Future Work

Semantics. The semantics of the rewrites are well defined in terms of the control flow graph. The control flow graph has a well defined relation to the semantics of the program. By formalising these relations we hope to develop general methods for establishing certain properties of the transformations we express, such as soundness or when the transformation is performance improving.

Annotated/Active Libraries. The example transformations presented in this paper are general ones to be applied to any program. These are well known and implemented in many compilers. However, code that is specialised to some specific purpose or particular architecture can be particularly amenable to certain specialised optimisations. To implement these optimisations automatically in “traditional” compilers involves detailed knowledge of the compiler design and implementation which is not available to everyone. Active libraries [31] try and bridge the gap between the optimising compiler and the library writer. The idea is that the library is annotated with domain specific information to help the compiler perform optimisations. Engler and his team at Stanford have explored the idea in some depth, and illustrated the possibilities in on wide variety of examples [13,14].

The rewrite language we have presented seems an ideal language for expressing different optimisations as annotations to a library. We hope to experiment with different domains/architectures to see how useful it would be. A good starting point might be to compare our language of optimising annotations with that of the Broadway compiler constructed by Guyer and Lin [17,18,19]. That work shares our concern that optimisations should be specified in a simple declarative style.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1985.
2. A. Aiken, M. Fuhndrich, J. Foster, and Z. Su. A toolkit for constructing type- and constraint-based program analyses. In *Second International Workshop on Types in Compilation (TIC '98)*, March 1998.
3. A. W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.
4. U. Assmann. How to uniformly specify program analysis and transformation with graph rewrite systems. In P. Fritzson, editor, *Compiler Construction 1996*, volume 1060 of *Lecture Notes in Computer Science*. Springer, 1996.

5. Uwe Aßmann. On Edge Addition Rewrite Systems and Their Relevance to Program Analysis. In J. Cuny, H. Ehrig, G. Engels, and G. Rozenberg, editors, *5th Int. Workshop on Graph Grammars and Their Application To Computer Science, Williamsburg*, volume 1073 of *Lecture Notes in Computer Science*, pages 321–335, Heidelberg, November 1994. Springer.
6. Uwe Aßmann. OPTIMIX, A Tool for Rewriting and Optimizing Programs. In *Graph Grammar Handbook, Vol. II*. Chapman-Hall, 1999.
7. A. J. C. Bik, P. J. Brinkhaus, P. M. W. Knijnenburg, and H. A. G. Wijshoff. Transformation mechanisms in mt1. Technical report, Leiden Institute of Advanced Computer Science, 1998.
8. J. M. Boyle. A transformational component for programming languages grammar. Technical Report ANL-7690, Argonne National Laboratory, IL, 1970.
9. J. M. Boyle. Abstract programming and program transformation. In *Software Reusability Volume 1*, pages 361–413. Addison-Wesley, 1989.
10. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8:244–263, 1996.
11. J. R. Cordy, I. H. Carmichael, and R. Halliday. The TXL programming language, version 8. Legasys Corporation, April 1995.
12. Steven Dawson, C. R. Ramakrishnan, and David S. Warren. Practical program analysis using general purpose logic programming systems — A case study. *ACM SIGPLAN Notices*, 31(5):117–126, May 1996.
13. D. R. Engler. Incorporating application semantics and control into compilation. In *Proceedings of the First Conference on Domain-Specific Languages*, pages 103–118. USENIX, 1987.
14. D. R. Engler. Interface compilation: Steps toward compiling program interfaces as languages. *IEEE Transactions on Software Engineering*, 25(3):387–400, 1999.
15. R. E. Faith, L. S. Nyland, and J. F. Prins. KHEPERA: A system for rapid implementation of domain-specific languages. In *Proceedings USENIX Conference on Domain-Specific Languages*, pages 243–255, 1997.
16. R. Giegerich, U. Moncke, and R. Wilhelm. Invariance of approximative semantics with respect to program transformations, 1981.
17. S. Z. Guyer and C. Lin. An annotation language for optimizing software libraries. In *Second conference on Domain-Specific Languages*, pages 39–52. USENIX, 199.
18. S. Z. Guyer and C. Lin. Broadway: A software architecture for scientific computing. Proceedings of the IFIPS Working Group 2.5 Working Conference on Software Architectures for Scientific Computing Applications. (to appear) October, 2000., 2000.
19. S. Z. Guyer and C. Lin. Optimizing high performance software libraries. In *Proceedings of the 13th International Workshop on Languages and Compilers for Parallel Computing. August, 2000.*, 2000.
20. R. Heckmann. A functional language for the specification of complex tree transformations. In *ESOP '88, Lecture Notes in Computer Science*. Springer-Verlag, 1988.
21. R. Johnson, D. Pearson, and K. Pingali. Finding regions fast: Single entry single exit and control regions in linear time, 1993.
22. M. Klein, J. Knoop, D. Koschützski, and B. Steffen. DFA & OPT-METAFrame: a toolkit for program analysis and optimization. In *Proceedings of the 2nd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '96)*, volume 1055 of *Lecture Notes in Computer Science*, pages 418–421. Springer, 1996.

23. J. Knoop, O. Rüthing, and B. Steffen. Towards a tool kit for the automatic generation of interprocedural data flow analyses. *Journal of Programming Languages*, 4:211–246, 1996.
24. P. Lipps, U. Mönke, and R. Wilhelm. OPTRAN – a language/system for the specification of program transformations: system overview and experiences. In *Proceedings 2nd Workshop on Compiler Compilers and High Speed Compilation*, volume 371 of *Lecture Notes in Computer Science*, pages 52–65, 1988.
25. Florian Martin. PAG – an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67, 1998.
26. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
27. R. Paige. Viewing a program transformation system at work. In *Proceedings Programming Language Implementation and Logic Programming (PLILP), and Algebraic and Logic Programming (ALP)*, volume 844 of *Lecture Notes in Computer Science*, pages 5–24. Springer, 1994.
28. B. Steffen. Data flow analysis as model checking. In *Proceedings of Theoretical Aspects of Computer Science*, pages 346–364, 1991.
29. B. Steffen. Generating data flow analysis algorithms from modal specifications. *Science of Computer Programming*, 21:115–139, 1993.
30. S. W. K. Tjiang and J. L. Henessy. Sharlit — a tool for building optimizers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1992.
31. Todd L. Veldhuizen and Dennis Gannon. Active libraries: Rethinking the roles of compilers and libraries. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98)*. SIAM Press, 1998.
32. E. Visser, Z. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In *International Conference on Functional Programming '98*, ACM SigPlan, pages 13–26. ACM Press, 1998.
33. D. Whitfield and M. L. Soffa. An approach for exploring code-improving transformations. *ACM Transactions on Programming Languages and Systems*, 19(6):1053–1084, 1997.

Compiler Transformation of Pointers to Explicit Array Accesses in DSP Applications

Björn Franke and Michael O’Boyle

Institute for Computing Systems Architecture (ICSA)
Division of Informatics
University of Edinburgh

Abstract. Efficient implementation of DSP applications is critical for embedded systems. However, current applications written in C, make extensive use of pointer arithmetic making compiler analysis and optimisation difficult. This paper presents a method for conversion of a restricted class of pointer-based memory accesses typically found in DSP codes into array accesses with explicit index functions. C programs with pointer accesses to array elements, data independent pointer arithmetic and structured loops can be converted into semantically equivalent representations with explicit array accesses. This technique has been applied to several DSPstone benchmarks on three different processors where initial results show that this technique can give on average a 11.95 % reduction in execution time after transforming pointer-based array accesses into explicit array accesses.

1 Introduction

Embedded processors now account for the vast majority of shipped processors due to the exponential demand in commodity products ranging from cell-phones to power-control systems. Such processors are typically responsible for running digital signal processing (DSP) applications where performance is critical. This demand for performance has led to the development of specialised architectures hand-coded in assembly. More recently as the cost of developing an embedded system becomes dominated by algorithm and software development, there has been a move towards the use of high level programming languages, in particular C, and optimising compilers. As in other areas of computing, programming in C is much less time consuming than hand-coded assembler but this comes at a price of a less efficient implementation due to the inability for current compiler technology to match hand-coded implementations.

To balance the requirement of absolute performance against program development time, there has been a move to tuning C programs at the source level. Although these tuned programs may perform well with the contemporary compiler technology for irregular DSP architectures [6], such program tuning frequently makes matters *worse* for optimising compilers for modern DSPs with large, homogeneous register sets and regular architectures. In particular, DSP

Example 1.1 Original pointer-based array traversal

```

int *p_a = &A[0] ;
int *p_b = &B[0] ;
int *p_c = &C[0] ;

for (k = 0 ; k < Z ; k++)
{
    p_a = &A[0] ;
    for (i = 0 ; i < X; i++)
    {
        p_b = &B[k*Y] ;
        *p_c = *p_a++ * *p_b++ ;
        for (f = 0 ; f < Y-2; f++)
            *p_c += *p_a++ * *p_b++ ;
        *p_c++ += *p_a++ * *p_b++ ;
    }
}

```

applications make extensive use of pointer arithmetic as can be seen in the DSPstone Benchmarks [13]. Furthermore in [10] programmers are actively encouraged to use pointer based code in the mistaken belief that the compiler will generate better code.

This paper is concerned with changing pointer based programs typically found in DSP applications into an array based form amenable to current compiler analysis. In a sense we are reverse engineering “dusty desk” DSP applications for modern high-performance DSPs.

In the next section we provide a motivating example using a typical DSP program and how our technique may transform it into a more efficient form. Section 3 describes the general algorithm and is followed in section 4 by an evaluation of our technique on a set of benchmarks from the DSPstone suite across three platforms. Section 5 describes related work in this area and is followed in section 6 by some concluding remarks.

2 Motivation

Pointer accesses to array data frequently occur in typical DSP programs that are tuned for DSPs with heterogeneous register sets and irregular data-paths. Many DSP architectures have specialised *Address Generation Units (AGUs)* [5], but early compilers were unable to generate efficient code for them, especially in programs containing explicit array references. Programmers, therefore, used pointer-based accesses and pointer arithmetic within their programs in order to give “hints” to the early compiler on how and when to use post-increment/decrement addressing modes available in AGUs. For instance, consider example [1.1] a kernel loop of the *DSPstone* benchmark `matrix2.c`. Here the pointer increment

Example 2.1 After conversion to explicit array accesses

```

for (k = 0 ; k < Z ; k++)
  for (i = 0 ; i < X; i++)
  {
    C[X*k+i] = A[Y*i] * B[Y*k];
    for (f = 0 ; f < Y-2; f++)
      C[X*k+i] +=
        A[Y*i+f+1] * B[Y*k+f+1];
    C[X*k+i] +=
      A[Y*i+Y-1] * B[Y*k+Y-1];
  }

```

accesses “encourage” the compiler to utilise the post-increment address modes of the AGU of a DSP.

If, however, further analysis and optimisation is needed before code generation, then such a formulation is problematic as such techniques often rely on explicit array index representations and cannot cope with pointer references. In order to maintain semantic correctness compilers use conservative optimisation strategies, i.e. many possible array access optimisations are not applied in the presence of pointers. Obviously, this limits the maximal performance of the produced code. It is highly desirable to overcome this drawback, without adversely affecting AGU utilisation.

Although general array access and pointer analysis are without further restrictions equally intractable [8], it is easier to find suitable restrictions of the array data dependence problem while keeping the resulting algorithm applicable to real-world programs. Furthermore, as array-based analysis is more mature than pointer-based analysis within available commercial compilers, programs containing arrays rather than pointers are more likely to be efficiently implemented. This paper develops a technique to collect information from pointer-based code in order to regenerate the original accesses with explicit indexes that are suitable for further analyses. Furthermore, this translation has been shown not to affect the performance of the AGU [25].

Example 2.1 shows the loop with explicit array indexes that is semantically equivalent to example 1.1. Not only it is easier to read and understand for a human reader, but it is amendable to existing array data-flow analyses, e.g. [3]. The data-flow information collected by these analyses can be used for e.g. redundant load/store eliminations, software-pipelining and loop parallelisation [14].

A further step towards regaining a high-level representation that can be analysed by existing formal methods is the application of de-linearisation methods. De-linearisation is the transformation of one-dimensional arrays and their accesses into other shapes, in particular, into multi-dimensional arrays [11]. The example 2.2 shows the example loop after application of clean-up conversion and de-linearisation. The arrays A, B and C are no longer linear arrays, but have been

transformed into matrices. Such a representation enables more aggressive compiler optimisations such as data layout optimisations [11]. Later phases in the compiler can easily linearise the arrays for the automatic generation of efficient memory accesses.

Example 2.2 Loop after pointer clean-up conversion and delinearisation

```

for (k = 0 ; k < Z ; k++)
  for (i = 0 ; i < X; i++)
  {
    C[k][i] = A[i][0] * B[k][0];
    for (f = 0 ; f < Y-2; f++)
      C[k][i] += A[i][f+1] * B[k][f+1];
    C[k][i] += A[i][Y-1] * B[k][Y-1];
  }

```

3 Algorithm

Pointer conversion is performed in two stages. Firstly, array and pointer manipulation information is gathered. Secondly, this information is used to replace pointer accesses by corresponding explicit array accesses, removing any pointer arithmetic.

3.1 Assumptions and Restrictions

The general problems of array dependence analysis and pointer analysis are intractable. After simplifying the problem by introducing certain restrictions, analysis might not only be possible but also efficient.

Pointer conversion may only be applied if the resulting index functions of all array accesses are affine functions. These functions must not depend on other variables apart from enclosing loop induction variables.

To facilitate pointer conversion, guarantee termination and correctness the overall affine requirement can be broken down further into the following restrictions:

1. structured loops
2. no pointer assignments except for initialisation to some array start element
3. no data dependent pointer arithmetic
4. no function calls that might change pointers itself
5. equal number of pointer increments in all branches of conditional statements

Structured loops are loops with a normalised iteration range going from the lower bound 0 to some constant upper bound N . The step is normalised to 1. Structured loops have the Single-Entry/Single-Exit property.

Pointer assignments apart from initialisations to some start element of the array to be traversed are not permitted. In particular, dynamic memory allocation and deallocation cannot be handled. Initialisations of pointers, however, to array elements may be performed repeatedly and may depend on a loop induction variable. Example 3.1 shows a program fragment with initialisations of the pointers `ptr1` and `ptr2`. Whereas `ptr1` is statically initialised, `ptr2` is initialised repeatedly within a loop construct and with dependence on the outer iteration variable `i`.

Example 3.1 Legal pointer initialisation / assignment

```
int array1[100], array2[100];
int *ptr1 = &array1[5];      /* OK */
int *ptr2;

for(i = 0; i < N; i++)
{
    ptr2 = &array2[i];        /* OK */
    for(j = 0; j < M; j++)
        ...
}
```

In example 3.2 illegal pointer assignments are shown. A block of memory is dynamically allocated and assigned to `ptr1`, whereas the initialisation of `ptr2` is runtime dependent. The assignment to `ptr3` is illegal if the pointer `ptrX` cannot be statically determined, but acceptable otherwise.

Example 3.2 Illegal pointer initialisations

```
int *ptr1, *ptr2, *ptr3;

ptr1 = (int *) malloc(...);  /* Dynamic memory allocation */
ptr2 = &a[b[i]];              /* b[i] data dependent */
ptr3 = ptrX;                  /* illegal if ptrX unknown */
```

Data dependent pointer arithmetic is the modification of a pointer (not the value pointed to) in a runtime dependent manner. Although there are powerful methods available e.g. [12] for this problem of pointer or alias analysis, such program constructs are not considered by our algorithm. Pointer expressions that can be statically determined are admissible (see Example 3.3)

Function calls might also modify pointers. Those functions that take pointers to pointers as arguments are disallowed as the actual pointers passed to the

Example 3.3 Data dependent and independent pointer arithmetic

```
ptr++;      /* Data independent */
ptr += 4;   /* Data independent */

ptr += x;   /* Dependent on x */
ptr -= f(y); /* Dependent on f(y) */
```

function itself and not only their content can be changed. Example 3.4 illustrates this point. The call to `function2` may possibly be handled using inter-procedural pointer analysis [7], but this is beyond the scope of this paper.

Example 3.4 Function calls changing pointer arguments

```
ptr = &array[0];

function1(ptr);    /* OK */

function2(&ptr);   /* not OK */
```

Finally, the number of increments of a pointer must be *equal in all branches* of conditional statements so that it is possible to statically guarantee that the value of a pointer at any point within the program, is the same regardless of runtime control-flow path. Situations with unequal number of pointer increments are extremely rare and typically not found in DSP code. The first if-statement in example 3.5 is legal, because `ptr1` is treated equally in both branches. In the second if-statement the increments are different, therefore this construct cannot be handled.

Note, however, that overlapping arrays and accesses to single arrays via several different pointers are perfectly acceptable. Because the conversion does not require information on such situations, but only performs a transformation of memory access representation, these kind of constructs that often prevent standard program analysis do not interfere with the conversion algorithm.

3.2 Overall Algorithm

During data acquisition in the first stage the algorithm traverses the *Control Flow Graph (CFG)* and collects information regarding when a pointer is given its initial reference to an array element and keeps track of all subsequent changes. Note that only changes of the pointer itself and not of the value of the object pointed to are traced. When loops are encountered, information regarding loop bounds and induction variables is recorded.

Example 3.5 Pointer increments in different branches

```

if (exp1)          /* Legal */
    x1 = ptr1++;
else
    y1 = ptr1++;

if (exp2)          /* Illegal */
    x2 = ptr2++;
else
    y2 = ptr2 + 2;

```

The main objective of the second phase is to replace pointer accesses to array elements by explicit array accesses with affine index functions. The mapping between pointers and arrays can be extracted from information gathered from pointer initialisation in the first phase. Array index functions outside loops are constant, whereas inside loops they are dependent on the loop induction variables. Information on pointer arithmetic collected during the first stage is then used to determine the coefficients of the index functions. Finally pointer-based array references are replaced by semantically equivalent explicit accesses, whereas expressions only serving the purpose of modifying pointers are deleted.

The pointer conversion algorithm can be applied to whole functions and can handle one- and multi-dimensional arrays, general loop nests of structured loops and several consecutive loops with code in between. It is therefore not restricted to handling single loops. Loop bodies can contain conditional control flow.

Algorithm [1](#) keeps a list of nodes to visit and depending on the type of statement, different actions will be performed. Pointer initialisations and assignments result in updates of the pointer-to-array mapping, whereas loops are handled by a separate procedure. Pointer arithmetic expressions are statically evaluated and pointer-based array references are replaced by equivalent explicit array references. The mapping between pointers contains not only the pointer and corresponding array, but also the initial offset of the pointer within the array and some local offset for keeping track of pointer increments in loop bodies.

The procedure for handling loops is part of the algorithm [1](#). Similar to the basic algorithm the loop handling procedure performs a pre-order traversal of the nodes of the loop body. Two passes over all nodes are made: The first pass counts the total offset within one loop iteration of pointers traversing arrays, the second pass then is the actual replacement phase. A final stage adjusts the pointer mapping to the situation after the end of the loop.

The algorithm passes every simple node once, and every node enclosed in a loop construct twice. Hence, the algorithm uses time $O(n)$ with n being the number of nodes of the CFG. Space complexity is linearly dependent on the number of different pointers used for accessing array elements, because for every pointer there is a separate entry in the *map* data structure.

Algorithm 1 Pointer clean-up conversion for CFG G **Procedure** clean-up(CFG G)

```

map  $\leftarrow \emptyset$ 
L  $\leftarrow$  preorderList( $G$ );
while L not empty do
  stmt  $\leftarrow$  head(L);
  removeHead(L);
  if stmt is pointer assignment statement then
    if (pointer,array,*,*)  $\in$  map then
      map  $\leftarrow$  map - (pointer,array,*,*)
    end if
    map  $\leftarrow$  map  $\cup$  (pointer,array,offset,0)
  else if stmt contains pointer reference then
    Look up (pointer,array,offset,*)  $\in$  map
    if stmt contains pointer-based array access then
      replace pointer-based access by array[initial index+offset]
    end if
    if stmt contains pointer arithmetic then
      map  $\leftarrow$  map - (pointer,array,offset,*)
      calculate new offset
      map  $\leftarrow$  map  $\cup$  (pointer,array,new offset,0)
    end if
  else if stmt is for loop then
    processLoop(stmt,map)
  end if
end while

```

Arrays Passed as Parameters of Functions. If no legal pointer assignment can be found, but the pointer used to traverse an array is a formal parameter of the function to be processed, it can be used as the name of the array [4].

Table 1. Absolute times for `biquad_N_sections` and the TriMedia

Benchmark	Level	TriMedia		Pentium II	
		Array	Pointer	Array	Pointer
biquad	-O0	11	9	1130	980
	-O1	11	8	490	360
	-O2	8	8	290	360
	-O3	9	8	360	350

Loop Nests. Perfect loop nests of structured loops are different from simple loops in the way that the effect of a pointer increment/decrement in the loop body not only multiplies by the iteration range of its immediately enclosing loop

Procedure processLoop(statement stmt, mapping map)

```

{count pointer increments in loop body and update map}
L = preorderList(loopBody)
while L not empty do
  stmt  $\leftarrow$  head(L);
  removeHead(L);
  if stmt contains array arithmetic then
    Update increment in (pointer,array,offset,increment)
  end if
end while

{replace pointer increments according to map}
L = preorderList(loopBody)
while L not empty do
  if stmt contains pointer reference then
    Look up (pointer,array,offset,increment)  $\in$  map
    Look up (pointer,local offset)  $\in$  offsetMap
    if (pointer,local offset)  $\notin$  offsetMap then
      offsetMap  $\leftarrow$  offsetMap  $\cup$  (pointer,0)
    end if
    if stmt contains pointer-based array access then
      index  $\leftarrow$  increment  $\times$  ind.var. + offset + local offset
      replace pointer-based access by array[index]
    end if
    if stmt contains pointer arithmetic then
      Update local offset in map
    end if
  end if
end while

{adjust all mappings to situation after end of loop}
for all (pointer,*,*,*)  $\in$  map do
  update offsets in map
end for

```

construct, but by the ranges of all outer loops. Therefore, all outer loops have to be considered when converting pointers of a perfect loop nest.

Handling perfect loop nests does not require extra passes over the loop. It is sufficient to detect perfect loop nest and descend to the inner loop body while keeping track of the outer loops. Once the actual loop body is reached conversion can be performed as usual, but with incorporating the additional outer loop variables and loop ranges as part of the index functions. Hence, asymptotical run-time complexity of the algorithm is not affected.

General loop nests can similarly be handled with a slightly extended version of the basic algorithm, by tracking which loop a pointer is dependent upon. The introductory example [\[1\]](#) illustrates a general loop nest and its conversion.

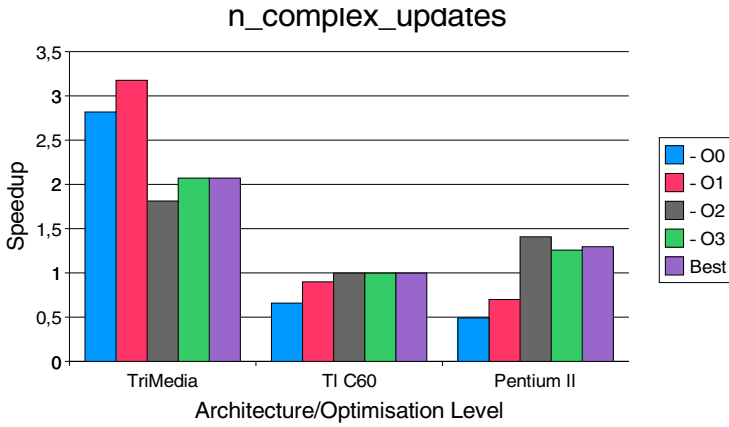


Fig. 1. Performance comparison of `n_complex_updates` benchmark

4 Experiments

The pointer clean-up conversion algorithm has been implemented as a prototype and integrated into the experimental source to source Octave compiler. After the application of the transformation on source-to-source level the resulting code is then used as an input for the C compilers of the Philips TriMedia TM-1 (compiler version 5.3.4), the Texas Instruments TMS320C6x (compiler version 1.10) and the Intel Pentium II (Linux 2.2.16, gcc version 2.95.2). Performance data was collected by executing the programs on two simulators (TM-1, TI C60) and a real machine (PII), respectively.

In order to quantify the benefit of applying the pointer clean-up conversion to DSP applications, program execution times for some programs of the DSPstone benchmark suite were determined. The speedups of the programs with explicit array accesses with respect to the pointer-based versions for the same optimisation level were calculated. As a DSP application developer typically wants the best performance we determined the optimisation level that gave the best execution time for the pointer and array codes separately. Due to the complex interaction between compiler and architecture, the highest optimisation level does not necessarily give the best performance. Thus the ratio between the best pointer-based version and the best explicit array access version is also presented.

Table 1 shows an example of the absolute performance figures for the `biquad_N_sections` benchmark on the TriMedia and Pentium II architectures that were the used for the computations of the speedup. Due to different optimi-

sations performed by the compilers at each optimisation level and different units of execution time, the reported times cannot be compared directly. Therefore, relative speedups based on the pointer-based program at the same optimisation level are shown in the following diagrams. As stated above, the minimal execution times are not necessarily achieved with the highest available optimisation level. Thus, it is reasonable to give an additional speedup measure when comparing the best pointer-based and explicit array based versions. For example, the shortest execution time achieved on the Pentium II was 350 for the pointer-based version and 290 for the explicit array access based version. Hence, the best speedup is $\frac{350}{290} = 1.21$. The figures 4 to 6 visualise the speedup measures for all three architectures and all optimisation levels.

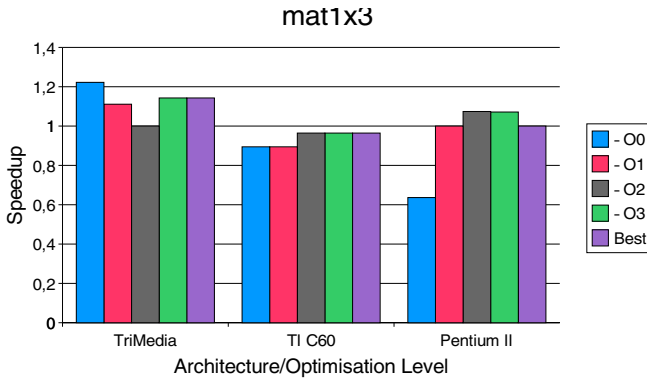


Fig. 2. Performance comparison of `mat1x3` benchmark

4.1 TriMedia

The performance of the TriMedia compiler/architecture combination usually benefits from using explicit array accesses rather than pointer-based array accesses. When there are no or only a few simple further optimisations applied (optimisation levels O 0 and O1), the program versions with explicit array accesses are significantly faster than the pointer-based programs. The difference can be up to 218% (`n_complex_updates`). For higher optimisation levels the differences become smaller. Although some of the original programs take less time than the transformed versions, in many cases there can be still some performance gain expected from pointer clean-up conversion. The pointer-based programs tend to

be superior at optimisations level O2, but the situation changes again for level O3 and the “best” case. This “best” case is of special interest since it compares the best performance of a pointer-based program to that of an explicit array access based program. The best explicit array based programs perform usually better or at least as well as the best pointer-based programs, only in one case (`dot.product`) a decrease in performance could be observed.

In figure 1 the speedup results for the `n_complex_updates` benchmark are shown. This rather more complex benchmark program shows the largest achieved speedups among all tests. The speedup reaches its maximum 3.18 at the optimisation level O1, and is still at 2.07 when comparing the best versions. This program contains a series of memory accesses that can be successfully analysed and optimised in the explicit array based version, but are not that easily amendable to simple pointer analyses. Although such large speedups are not common for all evaluated programs, it shows clearly the potential benefit of the pointer clean-up conversion.

Figure 2 compares the performances of the `mat1x3` benchmark. All speedups for the TriMedia architecture are ≥ 1 . The achieved speedups are more typical for the set of test programs. An increase of 14% in performance can be observed for the best explicit array based program version. With only a few other optimisations enabled the performance benefit of the transformed program is even higher.

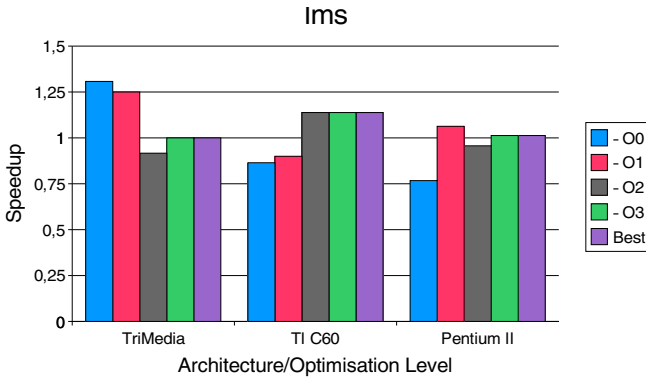


Fig. 3. Performance comparison of `lms` benchmark

The figures 3 and 4 represent the results of the `lms` and `matrix1` benchmark, respectively. As before, a significant speedup can be achieved at optimisation

levels O0 and O1. The performances of the explicit array accesses based versions are actually below those of the pointer-based versions at level O2, but at the highest level the transformed programs perform better or as well as the original versions. Although there is only a small speed up observable in these examples, it shows that the use of explicit array accesses does not cause any run-time penalty over pointer-based array traversals, but still provide a representation that is better suitable for array data dependence analysis.

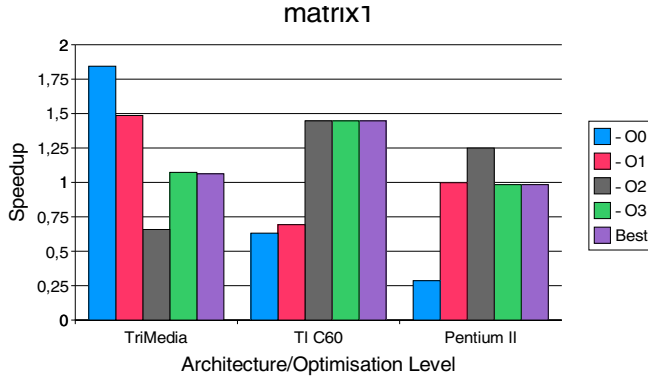


Fig. 4. Performance comparison of `matrix1` benchmark

In general, with pointer clean-up conversion some substantial benefit can be achieved without the need for further optimisations. On higher optimisation levels the transformed programs still perform better or at least as good as the pointer-based programs. Because all evaluated programs are not too complex, the original programs can often perform as good as the transformed programs at higher levels. But as shown in figure 4, for more complex programs the conversion provides some substantial advantage.

4.2 Pentium II

The Pentium II is a general-purpose processor with a super-scalar architecture. Additionally, it supports a CISC instruction set. This makes the Pentium II quite different from the TriMedia. However, many signal processing applications are run on Desktop PCs in which the Intel Processor is commonly found. Therefore, this processor is included in this evaluation.

The combination of the Intel Pentium II and the gcc compiler produces results that differ significantly from that of the TriMedia. The performance of

the program versions after pointer conversion is quite poor without any further optimisations, but as the optimisation level is increased the transformed programs often outperform the original versions. The maximum speedup of the transformed programs is usually achieved at optimisation levels O2 and O3, respectively.

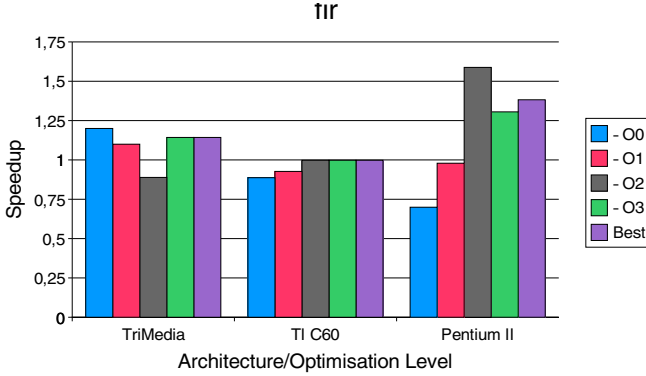


Fig. 5. Performance comparison of `fir` benchmark

In figure 5 the results for the `fir` benchmark are charted. Initially, the pointer transformation has negative effect on the performance of this program. A significant speedup can be observed for the higher optimisation levels. The maximum speedup achieved at level O2 is 1.59 and when comparing the two best versions each, the speedup of 1.38 can still be reached. The results of the `fir2dim` benchmark are represented in figure 6. As before the performance of the transformed version is inferior at optimisation levels O0 and O1, but it increases at O2 and O3. The performance can be increased by up to 6% in this case.

In general, the achievable speedups on the Pentium II architecture are not as large as they are on the TriMedia. It is not yet clear whether this is a consequence of architectural properties or the influence of the different compilers. However, an increase in the performance can be observed for the Intel processor, in particular at the higher optimisation levels.

4.3 TI TMS320C6x

The overall performance improvement of the Texas Instruments C60 lies between that of the Pentium II and the TM-1. In two cases the pointer conversion has no

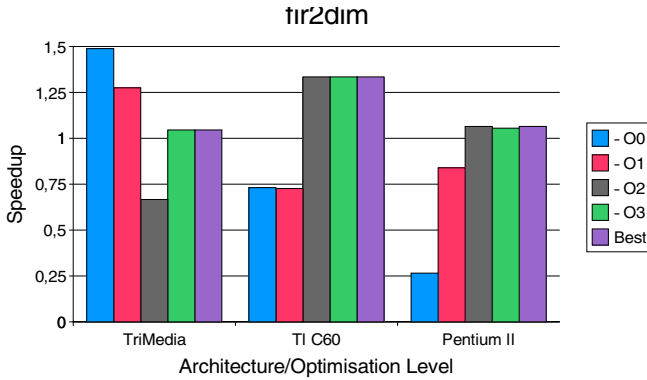


Fig. 6. Performance comparison of `fir2dim` benchmark

effect on the execution time of the programs, in one case there is actually a small degradation in performance, `mat1x3`, and in three cases there is a variability in terms of relative performance improvement with respect to the optimisation level. Increasing the optimisation level for the C60 improves the impact of the pointer conversion.

If we compare the behaviour of the two processors intended for embedded systems, we see that there is little correlation in their behaviour with respect to the conversion algorithm. The case where the TM-1 gains most from our technique, `n_complex_updates` has no impact on the C60; conversely the case where the C60 most benefits, `matrix1`, has little impact on the TM-1. From this we can conclude that the optimisation behaviour of both compilers is opaque and that further analysis is required to explain the behaviour in order to drive further optimisations.

5 Related Work

Previous work on transforming pointer accesses into a more suitable representation has focused on simplifying pointer expressions rather than transforming them into explicit array references. For example, Allan and Johnson [1] use their vectorisation and parallelisation framework based on C as an intermediate language for induction variable substitution. Pointer-based array accesses together with pointer arithmetic in loops are regarded as induction variables that can be converted into expressions directly dependent on the loop induction variable. The generated pointer expressions are more amenable to vectorisation than the

original representation, but this approach still does not fully regenerate the index expressions. For the main objective of vectorisation the method only treats loops individually rather than in a context of a whole function. The approach is based on a heuristic that mostly works efficiently, although backtracking is possible. Hence, in the worst case this solution is inefficient. No information is supplied about treatment of loop nests and multi-dimensional arrays. In his case study of the Intel Reference Compilers for the i386 Architecture Family Muchnick [9] mentions briefly some technique for regenerating array indexes from pointer-based array traversal. No more details including assumptions, restrictions or capabilities are given. The complementary conversion, i.e. from explicit array accesses to pointer-based accesses with simultaneous generation of optimal AGU code, has been studied by Leupers [5] and Araujo [2].

6 Conclusion

The contribution of this paper has been to introduce a new technique for transforming C code with pointer-based array accesses into explicit array accesses to support existing array data flow analyses and optimisations on DSP architectures. The approach has been implemented and integrated into the experimental Octave compiler and tested on examples of the DSPstone benchmark suite. Results show a significant 11.95 % reduction in execution time. The generation of efficient address generation code is improved since modern compilers are able to analyse array accesses and to generate optimised code for memory accesses that does not rely on the use of pointers at the source level.

We believe the use of explicit array accesses to be the key to automatic parallelisation of DSP applications for Multiprocessor DSP architectures. Future work will focus on distributing computation and data on different co-operating DSPs while making the best use of ILP and coarse-grain parallelism.

References

1. Allen R. and Johnson S., Compiling C for Vectorization, Parallelization, and Inline Expansion, *Proceedings of the SIGPLAN '88 Conference of Programming Languages Design and Implementation*, pp. 241-249, Atlanta, Georgia, June 22-24, 1988
2. de Araujo, Guido C.S., Code Generation Algorithms for Digital Signal Processors, Dissertation, Princeton University, Department of Electrical Engineering, June 1997.
3. Duesterwald E., Gupta R. and Soffa M., A Practical Data Flow Framework for Array Reference Analysis and its Use in Optimizations, *Proceedings of the SIGPLAN Conference on Programming Languages Design and Implementation*, 28(6), pp. 67-77, Albuquerque, New Mexico, 1993.
4. Kernighan, Brian W. and Ritchie, The C Programming Language, Second Edition, Prentice Hall, Englewood Cliffs, New Jersey, 1988.
5. Leupers R., Novel Code Optimization Techniques for DSPs, 2nd European DSP Education and Research Conference, Paris, France, 1998.

6. Liem C., Paulin P., Jerraya A., Address Calculation for Retargetable Compilation and Exploration of Instruction-Set Architectures Proceedings of the 33rd Design Automation Conference, Las Vegas, Nevada 1996.
7. Lu, J., Interprocedural Pointer Analysis for C, Ph.D. thesis, Department of Computer Science, Rice University, Houston, Texas, 1998. SASIMI, Osaka, 1997.
8. Maydan, Dror.E., John L. Hennessy, and Monica S. Lam., Effectiveness of Data Dependence Analysis, *International Journal of Parallel Programming*, 23(1):63-81, 1995.
9. Muchnick, Steven.S., Advanced Compiler Design and Implementation, Morgan Kaufmann Publishers, San Francisco, California, 1997.
10. Numerix-DSP Digital Signal Processing Web Site, http://www.numerix-dsp.com/c_coding.pdf, 2000.
11. O'Boyle M.F.P and Knijnenberg P.M.W., Integrating Loop and Data Transformations for Global Optimisation, *PACT '98, Parallel Architectures and Compiler Technology*, IEEE Press, October 1998.
12. Wilson, R.P., Efficient Context-Sensitive Pointer Analysis for C Programs, Ph.D. thesis, Stanford University, Computer Systems Laboratory, December 1997.
13. Zivojnovic, V., Velarde J.M., Schlager C. and Meyr H., DSPstone: A DSP-Oriented Benchmarking Methodology, *Proceedings of Signal Processing Applications & Technology*, Dallas 1994 .
14. Zima H., Supercompilers for Parallel and Vector Computers, ACM Press, 1991.

User-Extensible Simplification—Type-Based Optimizer Generators

Sibylle Schupp¹, Douglas Gregor¹, David Musser¹, and Shin-Ming Liu²

¹ Dept. of Computer Science, RPI {schupp,gregod,musser}@cs.rpi.edu

² Hewlett Packard shin@cup.hp.com

Abstract. For abstract data types (ADTs) there are many potential optimizations of code that current compilers are unable to perform. These optimizations either depend on the functional specification of the computational task performed through an ADT or on the semantics of the objects defined. In either case the abstract properties on which optimizations would have to be based cannot be automatically inferred by the compiler. In this paper our aim is to address this level-of-abstraction barrier by showing how a compiler can be organized so that it can make use of semantic information about an ADT at its natural abstract level, before type lowering, inlining, or other traditional compiler steps obliterate the chance. We present an extended case study of one component of a C++ compiler, the simplifier; discuss the design decisions of a new simplifier (simplifier generator) and its implementation in C++; and give performance measurements. The new simplifier is connected to the Gnu C++ compiler and currently performs optimizations at very high level in the front end. When tested with the Matrix Template Library, a library already highly fine-tuned by hand, we achieved run-time improvements of up to six percent.

1 Introduction

The introduction of abstract data types (ADTs) marks an important step for software production. By encapsulating implementation details of a type, abstract data types allow new separations of concerns and play a large role in making code sharing and large scale programming more feasible, especially after the extension of ADTs in the 1980s and 1990s to the object-oriented programming and generic programming styles. Although data abstraction is accepted in theory without reservation, it is fair to say that there is still considerable hesitation when it comes to using this style in practice. Ironically, the larger and more computationally intensive a project is, and the more it could take advantage of the safety and robustness an object-oriented approach provides, the stronger the opposition. The main concern articulated is efficiency, based on widespread experience that non-object-oriented programs outperform object-oriented ones, even when, in hybrid languages, features like dynamic method binding or inheritance are used cautiously. Depending on the programming language used, some simply accept overhead, while others resort to a programming style that anticipates and

then avoids the overhead of object-based programming. Thereby, however, they frequently sacrifice coding styles considerably. Especially in C++ there now exist high-performance libraries, e.g., BLITZ [33], the Parallel Object Oriented Methods and Applications project (POOMA) [24], and the Matrix Template Library (MTL) [28], that are able to compete with the efficiency of Fortran code, but pay for the performance gains with cryptic code at the user’s level. For example, a simple operator expression in these libraries is written as a function call with additional parameters or, even more unexpectedly from the user’s view, by a so-called expression template, a nest of class templates—constructs that allow library designers to hand-optimize away temporary variables and to thus overcome the “abstraction penalty.”

In this paper we address this level-of-abstraction barrier in a different way. We suggest that optimizations of ADTs take place during compilation and not in the source code. At the same time we suggest performing these optimizations directly at the level of ADTs. We claim that ADTs offer opportunities that are not available at a lower level of abstraction and would be obliterated by premature type lowering, inlining, or other traditional compiler steps. Since these optimizations either depend on the functional specification of the computational task performed through an ADT or on the semantics of the objects defined, and cannot be automatically inferred by a compiler, our approach is characterized by a flow of semantic information from the user program to the optimizer. While traditionally a compiler is a program that came into existence long before the user program, is applied to it as a black box, and delivers an executable without any communication with the program designer, a compiler in our understanding rather is a compilation generator that depends on (static) user information for its completion. With users extending the semantic knowledge available to the compiler, it can operate on strong assumptions about a type’s behavior and, as a result, generate more efficient code. Especially library designers, therefore, might find our approach useful: once they have provided the semantic information of a data type, its optimizations are available not only in all further compilations or linkings but also to all applications on top of the respective type. Thus, library clients get the benefit of efficient abstract data types in an entirely transparent way. When we use the term “user” in the following, we therefore mostly envision library designers and the like.

For the rest of the paper we focus on C++ [31]. Restricting ourselves to one component of a C++ optimizer, the simplification component, hence to the optimization of expressions, we show how to organize this component so that a normal C++ program can provide the semantic information needed to statically simplify an ADT in a correct and optimal way. In section 2 we summarize both the process of traditional simplification and the specification of the new simplifier. Key in decoupling the process of simplification and the semantics of data types is the *concept-based* approach, an approach that is increasingly used in the design of reusable software components, but also in compiler design in the related sense of *formal concept analysis*. In section 3 we briefly characterize programming with concepts, explain its advantages for the design of generative

optimizers, and illustrate concept-based simplification rules. We summarize the major design decisions behind the implementation in section 4 and illustrate our approach using an extended example of an algorithm of the MTL that is instantiated with a LiDIA [32] data type. In section 5 we demonstrate how the LiDIA user can guide the simplifier to replace expressions used in the MTL by more efficient LiDIA expressions—without changing the MTL code or compromising the readability of the source code. The statistical data of the run-time and code-size improvement of this and other experiments are reported in section 6. The approach of generative compilation seems to fit well in a recent trend of designing compilers that are more flexible or responsive. We compare our approach to others in more detail in section 7 and conclude the paper with a brief discussion of our future plans.

2 Simplification

We selected the simplification component because of its minimal dependencies on other components. In particular we wanted to avoid the need for any modifications to the data dependency analysis part of an optimizer. We will see, however, that the new approach to simplification not only extends the original functionality to user-defined types, but opens up new kinds of high-level optimizations. In the next subsections we summarize both traditional simplification and the features of our proposed generalization, including its benefits specifically for high-level optimizations.

2.1 Algebraic Simplification

A simplifier is an optimization component that operates on expressions. It rewrites a given expression (expression tree) to a different expression which is in some sense simpler: either as an expression tree of lower height or as an expression with cheaper operators, or as a combination of both. Its semantic knowledge base is a set of simplification rules that are subsequently matched against the expression under consideration; if no simplification rules applies, the expression is returned unchanged. A standard example of a simplification rule is

$$x + 0 \rightarrow x. \quad (*)$$

When this rule is applied, it replaces the addition by its first argument and thus saves carrying out the addition. Simplification rules can apply to more than one operator as in

$$(x \neq 0) \mid (y \neq 0) \rightarrow (x \mid y) \neq 0$$

and furthermore can be conditional:

$$\max(x, c) = x \quad \text{if } c \text{ is the smallest possible value.}$$

Since constant arguments like the 0 in the first simplification rule (*) rarely appear directly in user code, simplification traditionally takes place after other

parts of the optimizer have transformed a user-defined expression, possibly several times and from several perspectives. In the SGI MIPS and Pro64 compilers, for example, constant folding, constant propagation, array analysis, loop fusion, and several other loop transformations all have a subsequent simplification step that cleans up the previous results. Whether high- or low-level, however, simplification rules are phrased in terms of integral or floating-point expressions and, with a few exceptions of complex expressions, do not even apply to the classes of the C++ standard library, such as `string`, `bitmap`, `bool`, or the types of the Standard Template Library, much less to user-defined types.

2.2 User-Extensible Simplification

What are the characteristics of the proposed simplifier? First, it can be extended by arbitrary user-defined types. Admitting arbitrary types, however, gives rise to the question whether the current simplification rules and their operators, designed mainly for integers and floating-point numbers, are “substantial enough” to represent frequently occurring user-defined expressions that are worthwhile to optimize. Quite certainly, a numeric library uses different expressions with different frequencies than, say, a graphics package. If we admit new types but keep the simplification rules the same, the benefits of simplification might be severely restricted. Therefore, users should have the opportunity to extend the simplifier by their own operator (function) expressions as well as by simplification rules for those.

Next, all extensions should be easy to do. Although we can expect that users who are interested in the subtleties of performance issues are not programmer novices—as already mentioned, we envision library designers and the like as our special interest group—we certainly do not want them to have to modify the compiler source code. All extensions, in other words, should be done at the user’s level, through a user’s program. In section 3 we will explain how a concept-based approach helps to realize this requirement; here we only point out that all user-defined extensions are organized in header files and are compiled along with the main program. In the current implementation users have to manually include two header files—the predefined header `simplify.h` and the file with their own, application-specific simplifications—but we will automate the inclusion. All that’s left to do, then, is to turn on the simplifier with the optimization flag `-fsimplify`:

```
g++ -O3 -fsimplify prog.C
```

In summary, the new simplifier is characterized by the following properties:

- It is extensible by user-defined types, operators and function identifiers.
- It is extensible by user-defined simplification rules.
- All extensions are specified in ordinary user programs.
- It is organized so that the extensions do not entail any run-time overhead but are processed entirely at compile time.

Given a simplifier with these capabilities, users can then in fact go beyond the traditional, algebraic nature of simplification rules and use the simplification process to rewrite expressions in non-standardized ways. Thus far, we have come across two motivations for the introduction of non-algebraic rules: one is the separation of readability and efficiency at the source code level and the other is the possibility of rewriting certain expressions for the purpose of invoking specialized functions. In the first case simplification rules can be used to perform optimizations that otherwise take place in the source, e.g., of high-performance libraries. Rather than replacing operator expressions manually and at the user’s level by expression templates or other constructs that are efficient but hard to digest, library designers can introduce simplification rules that automatically rewrite expressions in the most optimal form. Thereby, neither readability nor efficiency is compromised. The other case occurs in the situation of parameterized libraries, where one library is instantiated with types of another, providing optimization opportunities the carrier library did not anticipate. Without the need to change this carrier’s code, simplification rules can be used here for replacing one function invocation by another. In both cases, the extensible simplifier allows library designers to pass optimizations on to the compiler that otherwise have to be done manually and directly at the source code level; for examples of both sets of rules see section 5.

3 Concept-Based Rules

Simplifying user-defined types, at first glance, might look like an impossible task because the types to be optimized are not known at simplifier design time. Without any knowledge of a type, its behavior or even its name, how could the simplifier treat it correctly and efficiently? However, the situation is similar to the situation in component-based programming where the clients and servers of a component also are unknown at component design-time—and yet the interaction with them has to be organized. In component-based programming there are several techniques available to model the necessary indirection between a component and its clients. The one that seems most appropriate for optimizer generators is the *concept-based* approach.

There are many definitions of “concept,” but as we use the term here, it means a set A of *abstractions* together with a set R of *requirements*, such that an abstraction is included in A if and only if it satisfies all of the requirements in R . This definition is derived in part from formal concept analysis [35,36], in which, however, the abstractions are usually merely names or simple descriptions of objects. Another source is the Tecton concept description language ([13,14]; see also [21]), in which the abstractions are more complex: they are algebras or other similar abstract objects such as abstract data types. An informal but widely known example of this understanding of concepts is the documentation of the C++ Standard Template Library developed at SGI [2]. There, the abstractions are types, and concepts like Assignable or Less Than Comparable are used to explicitly specify which requirements each type must satisfy in order to be

used as a specialization of a certain template parameter. In analogy, we state a simplification rule at a conceptual level, in terms of its logical or algebraic properties. While traditional simplification rules are directly tied to the types to which they apply, the concept-based approach allows referring, for example, to the rule (*) of section 2 abstractly as Right-identity simplification (fig. 1 lists more examples; for the representation of concepts in C++, see [26]).

left-id op X	$\rightarrow X$	$X \in M$	(M, op) monoid
X op right-inv(X)	\rightarrow right-id	$X \in G$	(G, op) group
$X = Y \cdot Z$	\rightarrow multiply(X, Y, Z)	X, Y, Z LiDIA::bigfloat	
$X = \text{bigfloat}(0)$	$\rightarrow X.\text{assign_zero}()$	X LiDIA::bigfloat	

Fig. 1. Concept-based rules (left) and the requirements for their parameters (right)

What are the advantages of the concept-based approach? Most importantly, concept-based rules allow for the decoupling of rules and types (type descriptors) that makes the extension of the simplifier possible. Since only requirements are mentioned and no particular types, the scope of a simplification is not limited to whichever finite set of types can be identified at simplifier design time. Instead, a rule can be applied to any type that meets its requirements. At the same time, the simplification code can be laid out at an abstract level and independently from particular types; it can contain the tests that check for the applicability of a rule and can symbolically perform the simplification. Secondly, concept-based rules help to reduce the amount of code of the compiler. The abstract Right-identity simplification rule, for example, corresponds to 16 concrete instances in the Pro64 SGI compiler, including $x + 0 \rightarrow x$, $x \cdot 1 \rightarrow x$, $x \&\&1 \rightarrow x$, or $\max(x, c) = x$ if c is the smallest possible value. Lifting several concrete rules to one abstract rule lowers the maintenance and debugging costs of the compiler itself. Lastly, abstractions makes user-extensions manageable. If users describe their types in the same abstract way that the simplifier states each of its rules, type descriptors can be provided independently from any particular rule and without any knowledge of the current set of simplifications. An internal query system can then check for superconcept/subconcept relations between the requirements of a rule and the behavior of a given type. The next section further discusses the implementation.

4 Implementation

The simplification framework is divided into three parts: the core simplifier or simplifier generator, which is the major part; an interface from a particular C++ compiler to the core simplifier; and the set of user-defined extensions (possibly empty). The core simplifier is currently connected to the Gnu compiler front end and can be used as-is with the Gnu and the Pro64 compilers, and others that use the Gnu front end. It is, however, a stand-alone program, which, if an appropriate

interface from the internal representation of the previous compilation step to the simplifier’s internal representation is provided, can work with any other C++ compiler that is standard-compliant and supports in particular all advanced template features.

Of particular importance for our purpose is the C++ feature of specialization, including partial specialization, where a set of class templates specialize in different ways one common class template (primary template, in C++ jargon). Originally, specialization was introduced to support different levels of genericity and different implementations of a generic task. The main advantage in our situation, however, is the fact that the compiler selects the most appropriate template based on the specializing type. Partial specialization is furthermore responsible for the Turing-completeness of the template sublanguage of C++—branching is modeled through template specialization, iteration through recursive templates—which allows us at least theoretically to implement simplification as a static program.

The key idea of the simplifier generator is in fact to express the generator entirely within the template sublanguage. Provided simplification rules are correctly represented as class templates, the normal instantiation mechanism of C++ and the partial order of partially specializing templates can then be used not only to select the best fitting simplification rule without additional effort on our part, but also to perform the selection at compile time. In addition, representing each rule as its own class, separated from all others, allows users to add new rules without modifying the existing ones. For this to work, however, we have to reorganize the formal side of a simplification rule as well as its actual instance. On the formal side, it is necessary that the selection of a simplification rule becomes statically decidable. While non-static simplification works on expressions directly, they now have to be modeled as types, so-called *expression templates*. With expressions as types, simplification rules can be represented as class templates so that the left-hand side of a rule constitutes the template interface and the right-hand side the template body. On the actual side of simplification expressions, consequently, we have to provide mappings between the representations that expressions have inside and outside the simplifier generator. Thus, the interface between the simplifier and the Gnu compiler essentially is concerned with the conversion from the Gnu’s internal TREE representation [4] to the expression-template representation used within the simplifier. Fig. 2 illustrates the template-based representation of a simplification rule using right-identity simplification. In this example, the class template is a partial specialization of the primary template `Simplify` by two parameters: an abstract binary expression, represented as expression template `BinaryExpr<BinaryOpClass, LeftOperand, RightOperand>`, and the set of constraints that a right-identity simplification imposes on the bindings of the three template parameters (`BinaryOpClass`, `LeftOperand`, `RightOperand`), encapsulated in the type `RightIdentitySimpl`. Together, the two parameters ensure that the simplification class is instantiated only when the bindings of all its parameters meet the requirements of a right-identity simplification. If they are met,

however, the class gets instantiated and the result of the simplification, that is, the right-hand side of the simplification rule, becomes accessible through the type definition in the body of the class. For the binding `BinaryExpr<Add,X,0>`, for example, (the type counterpart to the expression $x+0$) the type `Simplify::result` resolves to `X`, as expected.

```
template<class BinaryOpClass, class LeftOperand, class RightOperand>
struct Simplify<Expr<BinaryExpr<BinaryOpClass,LeftOperand,RightOperand>>,
               RightIdentitySimp>
{
    typedef typename Simplify<LeftOperand>::result result;
};
```

Fig. 2. Template representation of the right-identity simplification rule.

To summarize the technical details, the simplifier generator is based on advanced features of generic programming in C++, most notably the already mentioned expression templates [34][12], template metaprogramming [5], and traits (interface templates) [23]. As already mentioned, these features are increasingly used in C++ libraries, but can (or should, as we argued) be moved from the user level to the compiler level.

The set of concept-based rules is complemented by a set of conceptual type descriptors for each type that is the subject of simplification. This is the part where the simplifier is controlled by its users: they are expected to provide information about the logical, algebraic, or computational behavior of their types—whatever properties they omit, the simplifier will not be able to take advantage of. It might seem strange to hold users responsible for defining a type’s behavior but in some sense they only extend the responsibilities they already assume in traditional variable declarations. There, they assert their variable to be of a certain type; here, they additionally assert semantic properties. It also might appear that users have to provide a great amount of information or might even have to revise their type descriptors each time the set of simplification rules is extended. If stated abstractly enough, however, only a little information is necessary to integrate a user-defined type. For example, the left column in fig. 3 uses the LiDIA type `bigfloat` to show the complete, slightly idealized code to register this type with the simplifier. Extending the simplifier by new functions or expressions follows a similar scheme: each property the simplifier cannot infer has to be specified. Like type descriptors, the descriptors of functional expressions are standardized and implemented as so-called interface templates (or traits). The right column in fig. 3 gives two examples of traits defining the LiDIA member functions `is_zero` and `assign_zero`. The complete code of all LiDIA-specific extensions can be found in [10].

Concept-based rules and concept-based type (or function) descriptors, finally, are brought together in an internal validation step that checks for each simplification rule whether its semantic constraints are met. Given an actual expression

```

// Algebraic behavior
template<>
struct AlgebraTraits<bigfloat>
{
    typedef Field<bigfloat,
                Add, Mult, 0,
                UnaryMinus, Sub, 1,
                Recip, Div>
                structure;
};

// Computational behavior
template<>
struct TypeTraits<bigfloat>
{
    typedef __true_type is_applicative;
    typedef __true_type is_floating;
};

// Literal table
template<> bigfloat
LiteralTable<bigfloat>::literals[] =
{
    bigfloat(0),
    bigfloat(1)
};

struct UnaryOpTraits<
    LiDIA::EqualsZero, bigfloat>
{
    typedef __true_type is_applicative;
    typedef __false_type has_side_effects;
    typedef __true_type operand_is_const;
    typedef __false_type can_overflow;
    typedef __false_type can_underflow;
    typedef __false_type can_zero_divide;
    typedef bool result_type;
    static inline bool
    apply(const bigfloat& operand)
    {
        return operand.is_zero();
    }
};

struct UnaryOpTraits<
    LiDIA::AssignZero, bigfloat>
{
    typedef __false_type is_applicative;
    typedef __true_type has_side_effects;
    // ...
    static inline void
    apply(bigfloat& operand)
    {
        return operand.assign_zero();
    }
};

```

Fig. 3. Left: user-provided descriptors of LiDIA’s type `bigfloat` (complete, but slightly simplified). Right: user-defined expression descriptors of the LiDIA functions `is_zero` and `assign_zero`.

with an actual type the validator retrieves its type descriptor and compares the type specification against the requirements of the corresponding variable of the simplification rule that best fits the given expression. The comparison of requirements could be as simple as a check for equality, but often implies in practice a more complicated confirmation of one concept as a subconcept of another. To derive such relations the validator then searches a small internal repository. Suppose, for example, the rule (*) for right-identity simplification is about to be applied to an instance of, say, LiDIA’s `bigfloat` type. The validator first retrieves the requirements of the rule (which are specified for elements of monoid structures) and the description of the type (which forms a field), then performs a series of lookups in the internal repository to confirm the concept of a field as a subconcept of the monoid concept, and finally gives way to the instantiation of the simplification rule with the `bigfloat`-expression; as we have seen earlier

(fig. 2) the body of this class then holds the result of the symbolically simplified expression.

5 Extended Example

To demonstrate the optimization opportunities that an extensible simplifier provides, we now discuss an extended example using the libraries MTL (Matrix Template Library) [28] and LiDIA [32], a library for computational number theory.

<pre> T a = a.in, b = b.in; if (b == T(0)) { c_ = T(1); s_ = T(0); r_ = a; } else if (a == T(0)) { c_ = T(0); s_ = sign(b); r_ = b; } else { // cs= a / sqrt(a ^2+ b ^2) // sn=sign(a).b / sqrt(a ^2+ b ^2) T abs_a = MTL_ABS(a); T abs_b = MTL_ABS(b); if (abs_a > abs_b) { // 1/cs = sqrt(1+ b ^2/ a ^2) T t = abs_b / abs_a; T tt = sqrt(T(1) + t * t); c_ = T(1) / tt; s_ = t * c_; r_ = a * tt; } else { // 1/sn=sign(a). sqrt(1+ a ^2/ b ^2) T t = abs_a / abs_b; T tt = sqrt(T(1) + t * t); s_ = sign(a) / tt; c_ = t * s_; r_ = b * tt; } } </pre>	<pre> T a = a.in, b = b.in; if (b.is_zero()) { c.assign_one(); s.assign_zero(); r_ = a; } else if (a.is_zero()) { c.assign_zero(); s_ = sign(b); r_ = b; } else { // (see left column) // T abs_a = MTL_ABS(a); T abs_b = MTL_ABS(b); if (abs_a > abs_b) { // (see left column) T t = abs_b / abs_a; T tt = sqrt(T(1)+square(t)); (*) inverse(c_,tt); multiply(s_,t,c_); multiply(r_,a,tt); } else { // (see left column) T t = abs_a / abs_b; T tt = sqrt(T(1)+square(t)); (*) divide(s_,sign(a),tt); multiply(c_,t,s_); multiply(r_,b,tt); } } </pre>
--	--

Fig. 4. The body of the MTL algorithm `givens_rotation` with `T` bound to the LiDIA `bigfloat` type: the original MTL code (left) and the code generated by the simplifier (right). Numbers in parentheses at the end of a line indicate the number of saved temporary variables of type `bigfloat`; (*) marks rewrites by more efficient functions.

In short, MTL provides the functionality of a linear algebra package, but also serves as basis for sophisticated iterative solvers. It provides extensive sup-

port for its major data types, vectors and matrices, but relies on C++ built-in types for the element type of a matrix. LiDIA, on the other hand, provides a collection of various multi-precision types, mainly for arithmetic (e.g., in number fields) and cryptography. For the example we selected the MTL algorithm `givens_rotation`, performing the QR decomposition of the same name, and instantiated it with vectors over the LiDIA type `bigfloat`. The core of the function `givens_rotation` is listed in fig. 5. As the code listing (left column) shows there are several operator expressions that, when applied to instances of class types, require constructor calls, hence temporary variables. At the same time LiDIA’s `bigfloat` class has member functions defined that are equivalent to, but more efficient than some of these operator expressions: the application of the member function `is_zero`, for example, requires one temporary less than the equivalent operator expression `b == T(0)` (first line) and, likewise, the call `multiply(r_,b,tt)` is more efficient than `r_ = b*tt` (last line). In a traditional compilation model, LiDIA users would have to either accept performance losses or would have to modify MTL code; with the extensible simplifier, however, they can leave the MTL source code unchanged, but “overwrite” the generated code.

Using the techniques described in the last section they can register the `bigfloat` type and then specify which operator expressions they want to see replaced. We showed in fig. 3 how to instruct the simplifier so that it replaces test-for-zero expressions on `bigfloat` variables (`EqualsZero`) by calls to their member function `is_zero`. Following this scheme, we added altogether 6 LiDIA-specific rules for the example. These 6 rules, along with the general-purpose rules already in the simplifier, result in the saving of 12 temporaries (see fig. 5, right column); more precisely, the saving of constructing and destructing 12 `bigfloat` instances. The next section reports on the speed-up and code size saving gained.

6 Experimental Results

We conducted a series of experiments where we instantiated different MTL algorithms with LiDIA data types. For the presentation of the results in this section we selected three MTL algorithms in addition to the already discussed Givens rotation: the Householder QR transformation `generate_householder`, `_tri_solve` for solving triangular equation systems, and `_major_norm` for computing the 1- and ∞ -norms. All four routines are core routines for linear algebra and numerical computations. Another, more practical reason for selecting them was their code size in the MTL implementation, which, between 25 and 60 lines, seems to be large enough to allow for noticeable simplification effects. In the tests the simplifier was extended by the LiDIA-specific rules listed in figure 5.

MTL provides small demonstration programs for each algorithm that just generate its input arguments, run the algorithm, and print the result. We used these programs, but inserted cycle counters around the algorithm invocation. For the compilation we extended the Gnu compiler g++ 2.96 by an interface file to the simplifier. The version 2.96 of g++ has a well-known, serious performance bug in its inliner, which results in smaller performance improvements in three

$X = \text{bigfloat}(0) \rightarrow X.\text{assign_zero}()$	$X == \text{bigfloat}(0) \rightarrow X.\text{is_zero}()$
$X = \text{bigfloat}(1) \rightarrow X.\text{assign_one}()$	$X == \text{bigfloat}(1) \rightarrow X.\text{is_one}()$
$X * X \rightarrow \text{square}(X)$	$X = X \text{ op } Y \rightarrow X \text{ op} = Y, \text{ op} \in \{+, *\}$
$X = \text{op } Y \rightarrow \text{op}'(X, Y), \text{ op} \in \{1/-, \text{square}\}, \text{ op}' \in \{\text{invert}, \text{square}\}$	
$X = Y \text{ op } Z \rightarrow \text{op}'(X, Y, Z), \text{ op} \in \{+, -, *, /\}, \text{ op}' \in \{\text{add}, \text{subtract}, \text{multiply}, \text{divide}\}$	

Fig. 5. LiDIA-specific simplification rules

cases; in the `--tri.solve` test, we had to manually perform expression rewriting due to problems with the inliner interface, thus `--tri.solve` did not suffer from the same abstraction penalty losses as the other three cases. We then compiled each algorithm with and without the compilation flag `-fsimplify` at the highest optimization level, `-O3`, measured the size of the generated code and counted the cycles at run time. We also varied the precision of the floating point type, but this did not change a program’s behavior. To determine the number of cycles we ran each example 100,000 times, eliminated values that were falsified by interrupts, and computed the arithmetical means of the remaining values; each test was repeated several times. The tests were performed on a Linux platform with an AMD Duron processor, using MTL 2.1.2-19 and LiDIA 2.0.1.

The results show that we were able to achieve performance gains in all four cases, substantial gains for the Givens rotation (6%) and the `--tri.solve` algorithm (8%), and still 1% and almost 4% speed-ups in the `--major.norm` algorithm and the Householder transformation. It should be emphasized again both that manual rewriting changes the test conditions for `--tri.solve` and that the problems with the inliner of g++ are temporary and will be solved in its next release. With a fully working inliner without abstraction penalties we expect to further improve on the current performance gains. The speed-ups came along with a slight reduction of the code size in three cases, but the code size went up for the `major.norm` (by 1320 bytes). We suspect, again, the problem was due to the inliner.

Table 1 summarizes the changes to the code size and run time of each algorithm. It also lists for each algorithm its length in lines of code, the number of temporaries of type `bigfloat` that were saved, and, in parentheses, the total number of temporaries that could be saved with our approach but would have required extending the set of LiDIA-specific simplification rules.

7 Related Work

Our approach borrows from programming methodology and our motivation fits in a modern understanding of compilation; in this section we discuss related work in more detail. Programming with concepts, first, is an old approach underlying libraries in several languages, most notably Ada [22] and, with the most success, C++, where the Standard Template Library became part of the language specification [30,20]. The demand for adaptable components, at the same time,

Table 1. MTL algorithms simplified with LiDIA-specific simplification rules

Algorithm	Length (lines)	Temp. saved	Code size (bytes)			Run time (cycles)		
			before	after	savings	before	after	speedup
major_norm	28	3(5)	682216	683540	-1324	25704	25523	1.01 %
generate_householder	38	9	688037	687897	140	109343	105335	3.81 %
tri_solve	46	6	688775	688300	475	48095	44434	[8.24%]
givens_rotation	54	12	678450	678037	413	30112	28290	6.44 %

describes a recent trend in software development, which comes in several varieties. Although we were mostly inspired by generic programming in the sense of STL and its successors, the methodologies of adaptive and aspect-oriented programming [16] and intentional programming [29] bear resemblance to our goals but typically work neither with C++ nor with traditional (compilation) environments.

Extensible compilers are another recent trend, along with performance tuning for selected types and the introduction of features that allow advanced users to improve the optimizations their compilers perform. The ROSE source-to-source preprocessor [6], for example, extends C++ by a small generator for optimization specifications for class arrays. In the same direction, but not restricted to arrays, are the “annotation language for optimizing libraries” [11] and the OpenC++ project, part of the Open Implementation project at Xerox [3]. While the former can direct the Broadway compiler, the latter translates meta-objects in a C++ extension to source code that any C++ compiler can process. Particularly in C++, as already mentioned, much work is also done using the language itself, through traits, expression templates, and template metaprogramming, either to enforce certain (statement-level or cache) optimizations or to hand-code them [34,23,5,27]; see also [9] for a critical evaluation. Both ideas of modifying (extending) the translated language and modifying (customizing) the translated data types, however, are different from our approach of directly getting the optimization to work with arbitrary user-defined types. In performing type-based alias analysis, the latest implementation of the Gnu compiler realizes the advantages of higher programming constructs and type information for optimization tasks, but restricts type-based alias analysis to built-in types [18]; the Scale Compiler Group also reports the implementation of a type-based method for alias analysis [25]. Probably closest to our approach, except for the level of generality, comes the idea of *semantic expansion* in the Ninja (Numerically Intensive Java) project where selected types are treated as language primitives; for complex numbers and arrays the generated Java code outperforms both C++ and Fortran [19,37]. Finally, we want to point out that rewriting techniques have long been used in code generator generators, e.g., Twig [1], burg [8], and iburg [7].

8 Conclusions and Future Work

In conclusion, the new, extensible simplifier optimizes user-defined classes and has been successfully integrated into the Gnu C++ front end. It coexists with the original simplifier and lifts several old, concrete rules to an abstract level, but also contains rules without counterparts in the original simplifier. The examples with the LiDIA and MTL libraries show the substantial performance gains that are possible even at the level of very-high intermediate representations, and hopefully contribute toward overcoming the performance concerns of those who choose non-object-oriented C or Fortran programming over object-based programming in C++ or write code that compromises readability for efficiency. As pointed out in the introduction we consider the simplification project to be the start of a new approach to optimizer generators, which handle built-in types and classes equally. For the next steps in that direction we want to get more experience with other high-level optimizations, investigating in particular the required analytical parts. A promising immediate next step seems to be to extend the type-based (alias) analysis of the current Gnu compiler [18] to user-defined types.

Acknowledgments. We thank Sun Chan, Alex Stepanov, and Bolek Szymanski for their encouragement and help in defining a suitable focus for the project; Sun was also directly involved in the first stages of the simplifier project. We furthermore acknowledge Fred Chow for insights and contributions in the early discussions on higher type optimizations. This work was supported in part by SGI, Mountain View, CA.

References

1. A. V. Aho and S. C. Johnson. Optimal code generation for expression trees. *JACM*, 23(3):488–501, 1976.
2. M. Austern. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Addison-Wesley, 1999.
3. S. Chiba. A metaobject protocol for C++. In *Proc. of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 285–299, Oct. 1995.
4. CodeSourcery, LLC. *G++ Internal Representation*, August 2000.
<http://gcc.gnu.org/onlinedocs>
5. K. Czarnecki and U. W. Eisenecker. *Generative Programming—Towards a New Paradigm of Software Engineering*. Addison Wesley Longman, 2000.
6. K. Davis and D. Quinlan. ROSE II: An optimizing code transformer for C++ object-oriented array class libraries. In *Workshop on Parallel Object-Oriented Scientific Computing (POOSC'98)*, at *12th European Conference on Object-Oriented Programming (ECOOP'98)*, volume 1543 of *LNCS*. Springer Verlag, 1998.
7. C. W. Fraser, D. R. Hanson, and T. A. Proebsting. Engineering a simple, efficient code generator. *ACM TOPLAS*, 1(3):213–226, 1992.
8. C. W. Fraser, R. Henry, and T. A. Proebsting. BURG—fast optimal instruction selection and tree parsing. *SIGPLAN Notices*, 4(27):68–76, 1992.

9. D. Q. Frederico Bassetti, Kei Davis. C++ expression templates performance issues in scientific computing. In *12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing*, 1998.
10. D. P. Gregor, S. Schupp, and D. Musser. User-extensible simplification. A case study using MTL and LiDIA. Technical Report TR-00-7, Rensselaer Polytechnic Institute, 2000.
11. S. Z. Guyer and C. Li. An annotation language for optimizing software libraries. In T. Ball, editor, *2nd Conference on Domain-Specific Languages*. Usenix, 1999.
12. S. Haney, J. Crotinger, S. Karmesin, and S. Smith. PETE, the portable expression template engine. Technical Report LA-UR-99-777, Los Alamos National Laboratory, 1995.
13. D. Kapur and D. R. Musser. Tecton: A language for specifying generic system components. Technical Report 92-20, Rensselaer Polytechnic Institute Computer Science Department, July 1992.
14. D. Kapur, D. R. Musser, and X. Nie. An overview of the Tecton proof system. *Theoretical Computer Science*, 133:307–339, October 24 1994.
15. L.-Q. Lee, J. Siek, and A. Lumsdaine. The generic graph component library. In *Proc. of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA '99)*, volume 34, pages 399–414, 1999.
16. K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996. ISBN 0-534-94602-X.
17. S. B. Lippman. *C++ Gems*. Cambridge University Press, December 1996.
18. M. Mitchell. Type-based alias analysis. Dr.Dobb's Journal, October 2000.
19. J. E. Moreira, S. P. Midkiff, and M. Gupta. From flop to megaflops: Java for technical computing. *ACM TOPLAS*, 22(3):265–295, March 2000.
20. D. Musser and A. Saini. *STL Tutorial and Reference Guide. C++ Programming with the Standard Template Library*. Addison-Wesley, 1996.
21. D. Musser, S. Schupp, and R. Loos. Requirements-oriented programming. In M. Jazayeri, R. Loos, and D. Musser, editors, *Generic Programming—International Seminar, Dagstuhl Castle, Germany 1998, Selected Papers*, volume 1766 of *Lecture Notes in Computer Science*, Springer-Verlag, Heidelberg, Germany, 2000.
22. D. Musser and A. Stepanov. *The ADA Generic Library: Linear List Processing Packages*. Springer-Verlag, 1989.
23. N. Myers. A new and useful template technique. In *C++ Gems* [17].
24. J. V. Reynders, P. J. Hinker, J. C. Cummings, S. R. Atlas, S. Banerjee, W. F. Humphrey, S. R. Karmesin, K. Keahey, M. Srikant, and M. Tholburn. POOMA: A framework for scientific simulations on parallel architectures. In G. V. Wilson and P. Lu, editors, *Parallel Programming using C++*, pages 553–594. MIT Press, 1996.
25. Scale Compiler Group, Dept. of Comp. Science, Univ. Massachusetts. A scalable compiler for analytical experiments. <http://www-ali.cs.umass.edu/Scale>, 2000.
26. S. Schupp, D. P. Gregor, and D. Musser. Algebraic concepts represented in C++. Technical Report TR-00-8, Rensselaer Polytechnic Institute, 2000.
27. J. G. Siek. A modern framework for portable high performance numerical linear algebra. Master's thesis, Notre Dame, 1999.
28. J. G. Siek and A. Lumsdaine. The Matrix Template Library: A generic programming approach to high performance numerical linear algebra. In *International Symposium on Computing in Object-Oriented Parallel Environments*, 1998.

29. C. Simonyi. The future is intentional. *IEEE Computer*, 1999.
30. A. A. Stepanov and M. Lee. The Standard Template Library. Technical Report HP-94-93, Hewlett-Packard, 1995.
31. B. Stroustrup. *The C++ programming language, Third Edition*. Addison-Wesley, 3 edition, 1997.
32. The LiDIA Group. Lida—a C++ library for computational number theory. <http://www.informatik.tu-darmstadt.de/TI/LiDIA/>.
33. T. Veldhuizen. Blitz++. <http://oonumerics.org/blitz>
34. T. Veldhuizen. Expression templates. In *C++ Gems* [17].
35. R. Wille. Restructuring lattice theory: An approach based on hierarchies of concepts. In I. Rival, editor, *Ordered Sets*, pages 445–470. Reidel, Dordrecht-Boston, 1982.
36. R. Wille. Concept lattices and conceptual knowledge systems. *Computers and Mathematics with Applications*, 23:493–522, 1992.
37. P. Wu, S. P. Midkiff, J. E. Moreira, and M. Gupta. Efficient support for complex numbers in Java. In *Proceedings of the ACM Java Grande Conference*, 1999.

A Practical, Robust Method for Generating Variable Range Tables^{*}

Caroline Tice¹ and Susan L. Graham²

¹ Compaq Systems Research Center
caroline.tice@compaq.com

² University of California, Berkeley
graham@cs.berkeley.edu

Abstract. In optimized programs the location in which the current value of a single source variable may reside typically varies as the computation progresses. A debugger for optimized code needs to know all of the locations – both registers and memory addresses – in which a variable resides, and which locations are valid for which portions of the computation. Determining this information is known as the *data location problem*. Because optimizations frequently move variables around (between registers and memory or from one register to another) the compiler must build a table to keep track of this information. Such a table is known as a *variable range table*. Once a variable range table has been constructed, finding a variable's current location reduces to the simple task of looking up the appropriate entry in the table.

The difficulty lies in collecting the data for building the table. Previous methods for collecting this data depend on which optimizations the compiler performs and how those optimizations are implemented. In these methods the code for collecting the variable location data is distributed throughout the optimizer code, and is therefore easy to break and hard to fix. This paper presents a different approach. By taking advantage of *key instructions*, our approach allows the collection of all of the variable location data in a single dataflow-analysis pass over the program. This approach results in code for collecting the variable location information that is easier to maintain than previous approaches and that is almost entirely independent of which optimizations the compiler performs and of how the optimizations are implemented.

1 Introduction

A correct, accurate symbol table is critical for the interactive source-level debugging of optimized code. The symbol table contains information about all the

^{*} This work was supported in part by the Defense Advanced Research Projects Agency (DARPA) under Contract No. F30602-95-C-0136 and Grant No. MDA972-92-J-1028, by the National Science Foundation under Infrastructure Grant Nos. EIA-9802069 and CDA-9401156, by an endowment for the Chancellor's Professorship Award from the University of California, Berkeley, and by Fellowship support from the GAANN. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

symbols (names) that occur in the source program, including object names, type names, local and global variables, subroutine names and parameters. Symbol tables also keep information about the declared types of program variables and about how to map between locations in the source program and locations in the target program. An especially important piece of information kept in the symbol table is information about the places where variable values reside during program execution. Debuggers need accurate information about variable locations, both in memory and in registers, in order to determine where to look when a user asks to see the value of a variable during the execution of the program.

The relationship that exists between a source program and its corresponding *unoptimized* binary program makes it fairly straightforward to collect and generate symbol table information. For example, whenever the value of a source variable is updated, the new value is written to the variable’s home location in memory. Compiler optimizations destroy these relationships, making the task of determining a variable’s current location much harder. In an optimized program, the location that contains a variable’s correct current value can vary as the computation proceeds, sometimes being one of several different registers; sometimes being one of several locations in memory; sometimes being a constant value encoded directly in the instructions; and sometimes, during parts of the computation where the variable is not live, not being anywhere at all. Determining where to find a variable’s value is known as the *data location problem* [37].

In order to correctly capture information about variables’ locations, the compiler needs to construct a *variable range table*, such as the one described by Coutant, Meloy, and Ruscetta [3]. This table associates, with every variable in the source program, a list of “entries”. Each entry in the list gives a location in which that variable resides during some portion of the computation. Each location entry in the table also has associated with it the range(s) of addresses in the binary program for which the entry is valid.

The idea of a variable range table is not new; it has existed for over a decade. The difficulty is that, until now, there has been no easy way to collect this information. The standard approach is to track every source variable through every optimization performed by the compiler, recording each location as the variable gets moved around. Thus the data evolves as the optimizations are performed, and when the optimizations are complete the data has been “collected”.

This approach has many disadvantages. The greatest of these is that the code for collecting the variable range table information is distributed throughout the code for the optimizations. Consequently these implementations are very fragile, because any change to the optimizations (adding an optimization, removing an optimization, or changing the implementation of an optimization) can break this code. In addition errors in the code for collecting a variable location data are very hard to track down and fix, because the code is not all in one place.

In this paper we present a completely different technique for collecting this important information. By using dataflow analysis on the optimized binary program we are able to collect the variable location information in one pass. Our approach is independent of which optimizations the compiler performs and of

how the optimizations are implemented, although it does rely on accurate information about key instructions, as explained later. Our approach allows all the source code for collecting the location information to be in one place, rather than distributed throughout the optimizer source code in the compiler. This makes it simpler to find and fix errors in the location-collection code. The code is potentially more efficient than previous approaches, because the data is collected only once, not collected and then updated continually as optimizations are performed.

Our approach builds on the realization that the final optimized binary program must contain all the required information as to where variables are located. The optimized binary program encodes the final results of all the optimizations, including the low-level optimizations such as instruction scheduling, register allocation, and spilling. Therefore by performing dataflow analysis on the final optimized binary program and discovering where the location of a variable changes, we can obtain completely accurate and correct information as to the locations of all the program variables, only performing the analysis *once*, and without having to trace through each optimization individually.

So why has no one done this before? In order to perform the dataflow analysis on the instructions, one needs to know which instructions correspond to assignments to source program variables, and, for those instructions, which variable is receiving the assignment. As explained in the next section, this critical piece of information cannot be determined from the instructions by themselves. But we can take advantage of *key instructions* in order to obtain this vital information.

The rest of this paper is organized as follows: In Section 2 we describe our method for constructing the variable range table for an optimized program. We briefly describe our implementation experiences in Section 3, and in Section 4 we present our conclusions.

2 Using the Optimized Binary

The key to knowing where the current value of a variable resides at runtime is identifying the places in the target program where the value of a source variable gets stored, either because the variable has been copied to a new location, or because it has been given a new value. In this section, we explain how key instructions are used to find those places in the target program, and how dataflow analysis on those places is used to construct the variable range table.

2.1 The Role of Key Instructions

The notion of key instructions was originally introduced in order to solve a different problem relating to debugging optimized code, the *code location problem* [357]. This problem is determining how to map between locations in the source program and corresponding locations in the optimized target program. This mapping is needed for implementing many basic debugger functions, such as single-stepping and setting control breakpoints. The locations in the source program that are of interest are those that result in a source-level-visible state

change (i.e. changing either the value of a source variable or the flow of control through the program). Each piece of program source code (e.g., statement, expression, subexpression, etc.) that causes a single potential source-level-visible state change is called an *atom*. Some source statements contain only a single atom, but others contain multiple atoms. Every atom in the source program has a corresponding *key instruction* in the optimized target program.¹ Intuitively, the key instruction for any atom is the single instruction, in the set of instructions generated from that atom, that most closely embodies the semantics of the atom. In particular, of the set of instructions generated for the atom, the key instruction is the first instruction reached during program execution that causes a source-level-visible state change to the program. By definition every atom that has not been eliminated by the optimizer must have such an instruction.

Because we are concerned here with assignments to source variables, we focus on assignment atoms and their key instructions. Since there is a one-to-one mapping between atoms and visible state changes in the source, any atom can assign a value to at most one source variable.² Therefore the key instruction for an assignment atom is the instruction that writes the value of the right-hand side of the assignment to the variable’s location in memory; or, if the write to memory has been eliminated, it is the final instruction that evaluates the right-hand side into a register. If optimizations cause code for an atom to be duplicated, such as unrolling a loop some number of times, the atom will have one key instruction for each copy of the duplicated code that remains in the final program.

2.2 Internal Representation

<code>i = start - 1;</code>	<code>LOAD start; LOADI 1; SUB; STORE i;</code>
<code>while (!(done)) {</code>	<code>LOADI 0; LOAD done; WHILE NEQ;</code>
	<code>LOOP_BODY</code>
<code> i++;</code>	<code>LOAD i; LOADI 1; ADD; STORE i;</code>
<code> if (i >= j)</code>	<code>LOAD i; LOAD j; IF GEQ;</code>
	<code>TRUEBR</code>
<code> done = 1;</code>	<code>LOADI 1; STORE done;</code>
	<code>FALSEBR</code>
<code>}</code>	<code>END_LOOP</code>

(a) C code

(b) Early Internal Representation

Fig. 1. Sample C code and its equivalent early internal representation

Throughout this paper we refer to the *early internal representation* and to the *final internal representation* generated by the compiler. The *early internal*

¹ If code has been duplicated, a single atom may have multiple key instructions.

² For simplicity, this discussion assumes there are no constructs in the source, such as a swap statement, that simultaneously update multiple source variables.

representation is a very high-level set of instructions. While slightly “below” the level of the programming language (it uses explicit instructions for many things that are implicit in the programming language), it has constructs such as `if`, `while-do`, and `switch`. It also uses variable names to refer to variables that occur in the program source. Figure 1 shows a small piece of C code and how the code might be translated into this early internal representation. The early internal representation uses postfix operator conventions.

As the compilation proceeds this high-level representation gets gradually “lowered”. During this lowering process it is common for a single high-level instruction to be broken into multiple lower-level instructions. At the end of the compilation process the program is in the *final internal representation*, from which the binary code is directly generated. In the compiler we used, this internal representation is the assembly code for the RISC instruction set of the MIPS processor. Figure 2 shows some common instructions from this set.

Type	Example	Semantics
-----	-----	-----
Register Copy	or \$16, \$4, \$0	or \$4 with \$0 (always constant 0) and store in \$16. This is the same as to copying \$4 into \$16
	mov \$4, \$16	copy \$4 into \$16
Load Address	lda \$8, 24(\$sp)	\$8 gets the address ‘\$sp + 24’
Load	ld \$17, 32(\$sp)	\$17 gets contents of memory at location ‘\$sp + 32’
Store	sd \$31, 8(\$sp)	value in \$31 is written to memory at location ‘\$sp + 8’
Addition	addiu \$sp, \$sp, -80	\$sp gets the contents of \$sp added to -80 (an address)
	add \$16, \$4, \$11	\$16 gets the sum of the contents of \$4 and \$11
Subtraction	subiu \$sp, \$sp, 80	\$sp gets the contents of \$sp minus 80 (an address)
	sub \$7, \$8, \$5	\$7 gets the value of the contents of \$8 minus the contents of \$5

Fig. 2. Examples of instructions in the final internal representation

2.3 Identifying Key Instructions

We assume that the compiler keeps accurate information throughout the compilation process as to the source position (file, line, and column position) from which every piece of internal representation, both early and late, was generated. In the final representation, every instruction has associated with it the source position(s) from which it was generated. The compiler must maintain this basic information to make source-level debugging of the optimized program feasible.

Using the source position associated with each instruction, we can identify the set of all instructions generated from any given atom. Once we have this set for each atom we need to find the key instruction within the set. Identifying key instructions for control flow atoms is not too difficult: The key instruction for a control flow atom is the first conditional branching instruction in the set of instructions generated from that atom.³ Key instructions for assignment atoms, however, are much harder to identify. In fact they cannot be identified at all, if one has only the set of instructions generated from the atom.

There are three factors that contribute to this difficulty. First, one cannot identify assignment key instructions by the opcode of the instruction. An assignment key instruction might be a store instruction, but it could as easily be an add, subtract, multiply, or any other operator instruction (if the store to memory has been optimized away). Second, one cannot use variable names to determine the key instruction for an assignment, because there are no variable names in the instructions. Third, one cannot rely on the position of the instruction to identify assignment key instructions. While it is true that, in the absence of code duplication, the key instruction for an assignment atom will be the last non-nop instruction for the atom, optimizations such as loop unrolling may result in an assignment atom having multiple key instructions within a single basic block, making it impossible to use instruction position to find them all. In fact the only way to identify all the key instructions for an assignment atom is to perform a careful semantic analysis of both the instructions and the original source atom.

Luckily the compiler *does* perform such an analysis early in the compilation, when it parses the program and generates the early internal representation. It makes sense to take advantage of the compiler and have the front end tag the piece of early internal representation that will become the key instruction(s) for an assignment atom. These tags can then be propagated appropriately through the compiler as the representation is lowered, instructions are broken down, and optimizations are performed. When the final instructions are generated, the key instructions for assignment atoms are already tagged as such.

We claimed earlier that our approach to collecting variable location data is mostly independent of the optimization implementations. It is not completely independent because the key instruction tags have to be propagated through the various optimization phases. We feel this is acceptable for two reasons. First it is far simpler to propagate the one-bit tag indicating that a piece of internal representation corresponds to an assignment key instruction, than to track and record information about variable names and current variable storage locations. Second, and more importantly, we assume that key instructions will be tagged anyway, as a necessary part of solving the code location problem, without which one cannot implement such basic debugger functions as setting control breakpoints and single-stepping through the code. Therefore we view our use of key instructions to collect the variable range table information as exploiting an existing mechanism, rather than requiring a new, separate implementation.

³ By “first” we mean the first instruction encountered if one goes through the instructions in the order in which they will be executed.

For more information about key instructions, including formal definitions, algorithms for identifying them, and an explanation of how they solve the code location problem, see Tice and Graham [5].

2.4 Collecting Names of Assigned Variables

Recall that, in order to perform the dataflow analysis on the binary instructions, we need to know which instructions assign to source variables, and which variables receive those assignments. The key instruction tags on the instructions identify the assignment instructions, but we still need to know the name of the variable that receives each assignment. The phase to collect that information is executed fairly early in the back end of the compiler, before variable name information has been lost. In this phase a single pass is made over the early representation of the program. Every time an assignment is encountered, it is checked to see if the assignment is to a source-level variable.⁴ If so, the name of the variable and the source position of the assignment atom are recorded in a table. Thus at the end of this pass we have created a table with one entry for every assignment atom in the source program. Each entry contains the name of the variable receiving the assignment and the source position of the assignment atom. We attempt to make the variable name information collected in this phase as precise as possible, by recording assignments to components of variables such as fields or array elements.

2.5 Performing the Dataflow Analysis

A forward dataflow analysis is performed at the very end of the compilation process, after *all* optimizations (including instruction scheduling) have finished. The data items being created, killed, etc. by this dataflow analysis are *variable location records*, where a variable location record is a tuple consisting of a variable name, a location, a starting address, and an ending address. For the rest of this paper we will refer to such a tuple as a *location tuple*.

Location Tuple Format:

```
< name , location, starting address, ending address >
```

Example Location Tuples:

```
1: < 'x', $4, 0x1000A247, undefined >
2: < 'p.foo', Mem[$sp + $8], 0x1000A216, 0x1000A4BC >
```

Fig. 3. Location Tuples

⁴ The key instruction tags can be used to tell whether or not an assignment is to a source variable.

Figure 3 shows the format of a location tuple and gives two examples. Example 1 is a location tuple for variable x . It indicates that the value for x can be found in register four (\$4) while the program counter is between 0x1000A247 (the starting address for this tuple) and some undefined ending address. The ending address is undefined because the tuple shown in Example 1 is still live in the dataflow analysis. Once the tuple has been killed, the ending address will be filled in, as explained below. Example 2 is a tuple indicating the value for $p.foo$ can be found in the memory address resulting from adding the contents of register eight (\$8) to the stack pointer (\$sp) when the program counter is between 0x1000A216 and 0x1000A4BC.

When a location tuple is created, it receives the name of the source variable being updated or moved and the new current location for the variable (either a register or a memory address). The starting address is the address of the current instruction, and the ending address is undefined. When a location tuple is killed during the dataflow analysis, the ending address in the location tuple is filled in with the address of the killing instruction. Killed location tuples are not propagated further by the dataflow analysis, but they are kept. At the end of the dataflow analysis, all location tuples will have been killed. The data in these location tuples is then used to construct the variable range table.

We perform the dataflow analysis at the subroutine level. The initial set of location tuples for the dataflow analysis contains one location tuple for each formal parameter of the subroutine. (The compiler knows the initial locations for the parameters). It also contains one location tuple for each local variable to which the compiler has assigned a home location in memory. These local variable location tuples start with a special version of the variable name that indicates that the variable is uninitialized. When the variable is first assigned a value, a new location tuple is created for it. This allows us to determine those portions of the program where a variable is uninitialized, which in turn makes it easy for the debugger to warn a user who queries the value of an uninitialized variable.

In most respects the details of our dataflow algorithm are quite standard [2]. But the rules for creating and killing location tuples inside a basic block deserve comment. Although we have focused so far on instructions that update the values of source-level variables, in fact *every* instruction in the basic block must be examined carefully when performing the dataflow analysis. The following paragraphs enumerate the different types of instructions that require some kind of action and explain what actions are appropriate. Appendix A gives our algorithm for examining instructions within basic blocks.

In Figure 2 the reader can see all the various types of instructions discussed below except for key instructions. A key instruction is simply a regular instruction with an extra bit set.

Key Instructions. First we need to determine if the key instruction is for an assignment atom or not. We can determine this by comparing the source position associated with the key instruction with the source positions in the name table collected earlier (see Section 2.4). If the source position is not in the table, we know the key instruction is not for an assignment atom, so we treat it like any

other instruction. If the source position is found in the table, then we use the table to get the name of the variable receiving the assignment. We kill any other location tuples for that variable. We also kill any location tuple whose location is the same as the destination of the assignment instruction. Finally we generate a new location tuple for the variable, starting at the current address, and with a location corresponding to the destination of the assignment instruction.

Register Copy Instructions. If the instruction copies contents from one register to another, we first kill any location tuples whose locations correspond to the destination of the copy. Next we check to see if the source register is a location in any of the live location tuples. If so, we generate a new location tuple copying the variable name from the existing location tuple. We make the starting address of the new location tuple the current address, and set the location to the destination of the register copy. This can, of course, result in the same variable having multiple live location tuples, at least temporarily.

Load Instructions. For load instructions, we check the memory address of the load to see if it corresponds to any of the home addresses the compiler assigned to any local variables or to any location in a live location tuple (i.e. a spilled register). In either case we generate a new location tuple for the variable, starting at the current address, and using the destination of the load as the location. We also kill any existing live location tuples whose locations correspond to the destination of the load.

Store Instructions. First we kill any location tuples whose locations correspond to the destination of the store. Next we check the address of the store to see if it corresponds to an address the compiler assigned to a variable, in which case we generate a new location tuple for the variable. We also generate a new location tuple if the source of the store corresponds to a location in any live location tuple (i.e. a register spill).

Subroutine Calls. For subroutine calls we kill any location tuple whose location corresponds to subroutine return value locations.

All Other Instructions. If the instruction writes to a destination, we check to see if the destination corresponds to any location in any live location tuple. Any such location tuples are then killed. (The corresponding variable's value has been overwritten.)

2.6 Handling Pointer Variables

The actions outlined above do not take pointer variables into consideration. Pointers require slightly more work, but the basic ideas are fairly straightforward. In order to correctly gather and maintain location information for pointer variables, we take the following actions, in addition to those already described above. The following explanations use C syntax in particular, but the concepts and actions are generalizable for other languages.

Load Instructions. If the base register for a load from memory instruction corresponds to the location in any live location tuple T , and the load offset is zero, we generate a new live location tuple. The location in the new location tuple is the destination of the load instruction. The variable name in the new tuple,

V_1 , is a version of the variable name V that occurs in T , modified as follows. If V begins with at least one ampersand ('&'), we remove one ampersand from the front of V to construct V_1 . If V does not have at least one ampersand at the beginning, we add an asterisk ('*') to the front of V to construct V_1 .

Store Instructions. If the base register for a store to memory instruction corresponds to the location in any live location tuple T , and the store offset is zero, we generate a new location tuple. The location in the new tuple is "Memory[\$R]", where "\$R" is the base register of the store instruction. The variable name in the new tuple, V_1 , is a version of the variable name V that occurs in T , modified as follows. If V begins with an ampersand, remove one ampersand from the front of V to construct V_1 . Otherwise add one asterisk to the front of V to construct V_1 . In addition to creating the new location tuple, we also need to look for any live location tuples whose variable names could be constructed either by removing one or more ampersands from V or by adding one or more asterisks to V . Any such tuples must be killed, since the value stored in the location represented by our new location tuple will invalidate them.

Load Address Instructions. If the instruction set includes special instructions for loading addresses into registers, these instructions also require special actions. If the "base register + offset" used in the address calculation corresponds to a memory address in any live location tuple, or to the home location the compiler assigned to any local variable, we generate a new location tuple. The location in the new tuple is the destination of the load address instruction. The variable name is '&V', where V is the name of the variable whose home location or live location tuple location corresponds to the "base register + offset" used in the address calculation.

Integer Addition/Subtraction Instructions. If one operand of an integer addition or subtraction instruction is a register containing a memory address, treat the instruction exactly like the load address instructions, as explained above, using the other operand as the "offset". A register contains a memory address if it is one of the special registers that always address memory, such as \$sp or \$gp, or if it is the location in a live location tuple whose variable name begins with an ampersand.

2.7 Building the Variable Range Table

Once the dataflow analysis is complete, we have a large amount of data that needs to be combined into the variable range table. To make the range table as small as possible, we combine and eliminate location tuples wherever this is possible and reasonable. We also compute, for every variable, all the address ranges for which the variable does not have any location. Figure 4 traces an example of this process.

Figure 4(a) shows an example of raw location tuples collected for a variable x . In order to consolidate the data we first sort the location tuples by variable name. For each variable, the location tuples for that variable need to be sorted by location (Figure 4(b)). Any location tuples for a given variable and location that have consecutive ranges need to be combined. For example, the location tuples

<x, \$6, 64, 92>	<x, M[\$sp+48], 84, 100>	<x, M[\$sp+48], 84, 100>
<x, \$11, 24, 28>	<x, \$6, 64, 92>	<x, \$6, 64, 92>
<x, M[\$sp+48], 84, 100>	<x, \$6, 32, 48>	<x, \$6, 32, 48>
<x, \$11, 28, 36>	<x, \$11, 24, 28>	<x, \$11, 12, 36>
<x, \$6, 32, 48>	<x, \$11, 28, 36>	
<x, \$11, 12, 24>	<x, \$11, 12, 24>	
(a)	(b)	(c)
<x, \$11, 12, 36>	<x, \$11, 12, 32>	<x, uninit, 0, 12>
<x, \$6, 32, 48>	<x, \$6, 32, 48>	<x, \$11, 12, 32>
<x, \$6, 64, 92>	<x, \$6, 64, 84>	<x, \$6, 32, 48>
<x, M[\$sp+48], 84, 100>	<x, M[\$sp+48], 84, 100>	<x, evicted, 48, 64>
		<x, \$6, 64, 84>
		<x, M[\$sp+48], 84, 100>
(d)	(e)	(f)

Fig. 4. Trace for constructing variable range table information from raw location tuple data.

$\langle x, \$4, 1, 7 \rangle$ (translated as “variable x is in register 4 from address one through address seven”) and $\langle x, \$4, 7, 10 \rangle$ (“variable x is in register 4 from address seven through address ten”) can be combined into $\langle x, \$4, 1, 10 \rangle$ (“variable x is in register 4 from address one through address ten”). Figure 4(c) shows consecutive ranges for x in register 11 combined into a single tuple.

After all such consecutive location tuples have been combined, we sort all of the location tuples for each variable (regardless of location) by starting address (Figure 4(d)). For some address ranges a variable may be in multiple locations, while for other address ranges a variable may not be in any location. For those address ranges where a variable has multiple locations, we select one location (usually the one with the longest address range) to be the location we will record in the variable range table. We also shorten other ranges, to eliminate overlap. This is illustrated in Figure 4(e), where the ranges of x in registers $\$r11$ and $\$r6$ were shortened. This is an implementation decision, rather than a feature of our algorithm. Admittedly by not recording all locations for a variable we are losing some information. If the debugger for optimized code were to allow users to update the values of variables, then the information we are losing would be critical. We are assuming, for this work, that users are not allowed to update source variables from inside the debugger once a program has been optimized, as such an update could invalidate an assumption made by the compiler, which in turn could invalidate an optimization, and thus make the program start behaving incorrectly. Updating variables after optimizations have been performed is an open research issue, and is beyond the scope of this paper. For those address ranges for which a variable does not have any location, we create an “evicted”

record for the variable and put that in the variable range table.⁵ Thus an additional benefit of this approach is that we can automatically determine not only the correct locations of variables, but also the address ranges for which variables are uninitialized or evicted. Figure 4(f) shows the final set of location tuples for x , which will be written to the variable range table.

For location tuples whose variable names are of the form `array[exp]`, or which begin with one or more asterisks, we do not create any eviction records. Also for variables of the latter type, we do not keep location tuples whose location is of the form “Memory[exp]”, if there are also tuples, covering the same address ranges, whose location is “exp”, and whose name is constructed by removing all the asterisks from the front of the variables in question. For example, if we know that pointer variable p is in register sixteen, there is no point in keeping the information that $*p$ can be found in Memory[\$16]. On the other hand, knowing that $*p$ can also be found in register seven is useful, so such information is kept.

3 Implementation Experience

We implemented a prototype of our approach for building a variable range table inside the SGI Mips-Pro 7.2 C compiler, a commercial compiler that optimizes aggressively. This compiler consists of nearly 500,000 lines of C and C++ source code. The symbol table format used is the DWARF 2.0 standard format [4]. Since we were implementing an entire solution for debugging optimized code, we modified the compiler to identify key instructions as well as to generate the variable range table information. We also modified a version of the SGI dbx debugger to use the resulting symbol table for debugging optimized code.

Modifying the compiler to collect the variable names, to perform the dataflow analysis and to write the variable range table into the symbol table took roughly 2 months, and required modifying or writing approximately 1,500-2,000 lines of code. Modifying the debugger to use the variable range table to look up a variable’s value took about 2 weeks and 300-500 lines of code. As these numbers show, it is quite easy to adapt existing compilers and debuggers to create and use this variable range table. The numbers here do not include the time it took to implement the key instruction scheme, which took roughly five months and involved 1,500-2,000 lines of code. A large part of the five months was spent becoming familiar with the compiler code.

Once our prototype was fully implemented, we made it available for use to some software engineers at SGI and to some computer science graduate students at the University of California, Berkeley. In the end, sixteen people tested our prototype and gave us feedback on it. Overall the feedback was positive, several people expressing the wish they had had this tool a few months earlier, as it would have helped them find bugs they had been working on. For more details on our user feedback, see Tice [6].

⁵ Evicted variables are part of the *residency problem* identified by Adl-Tabatabai and Gross [1].

Prior to the implementation of our prototype, the SGI compiler, in common with most commercial compilers, did nothing in particular to facilitate debugging optimized code. Its default mode was to turn off optimizations if debugging was requested. Although it was possible to override this default through the use of certain compilation flags, the debugging information was collected and stored in the symbol table in exactly the same manner as if no optimizations were being performed. The result of this was that data in the symbol table for optimized programs was often incomplete, incorrect, and/or misleading, particularly with respect to the locations of variables.

When implementing our dataflow analysis we encountered a small problem. As in standard dataflow analysis algorithms, the in-set for each basic block was constructed by joining all out-sets of the predecessor basic blocks. Occasionally when joining sets of locations from predecessor blocks we found inconsistencies between the predecessors. Two different basic blocks might indicate the same variable being in two conflicting locations, as shown in Figure 5. At the end of basic block BB2, variable v is in register $\$r4$, while at the end of basic block BB3, v is in register $\$r7$.⁶ The question is where, at the beginning of basic block BB4, should we say that v is? The answer would appear to be “it depends on the execution path”. However the execution path is something we cannot know at compile time, when we are constructing the variable range table.

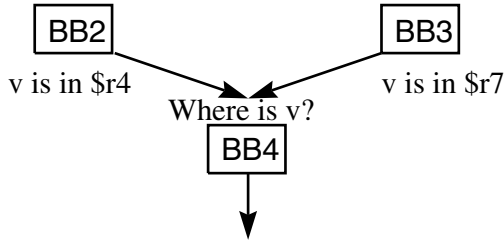


Fig. 5. Inconsistency problem during dataflow analysis

To deal with this problem, we chose to “kill” both of the conflicting location tuples at the end of the basic blocks from which they came. If there are no location tuples for the variable that do not conflict at this point, the range table will indicate the variable as being *evicted*, i.e. not residing anywhere. This choice was reasonable because the problem in Figure 5 typically arises only when v is dead on entry to BB4. Otherwise the compiler would require definitive information as

⁶ Note that it would be fine if BB2 and BB3 both reported that v was stored both in $\$r4$ and $\$r7$.

to where it could find the variable.⁷ Since the variable is dead, killing its location tuple (evicting it, in effect) seems a reasonable, simple solution.

An alternative approach would be to tag the conflicting location tuples in some special way. In the final variable range table all the conflicting alternative locations could be entered, leaving the correct resolution up to the debugger or the debugger user.

Our simple approach was particularly attractive in the context of Optdbx [6], the debugger for which it was built, because Optdbx performs eviction recovery, capturing the values of variables before the compiler overwrites them. This technique allows the debugger to recover the correct variable value in these cases. Using our example as illustration, whenever execution reaches either the end of basic block BB2 or basic block BB3, the debugger will notice that *v* is about to be evicted and so will cache the current value of *v* in a special table. If the user suspends execution at the top of basic block BB4 and asks to see the value of *v*, the debugger will first look in the range table, where it will see that *v* is currently evicted. It will then get the value of *v* from its special table.

The preceding discussion illustrates one of the added benefits of using dataflow analysis to construct the variable range table: one can obtain (with no extra work) exact information about variable evictions, uninitialized variables, and variables that have been optimized away. (The last case would be those variables for which no location tuples were created.) Users of our debugger especially liked its ability to indicate these special cases.

4 Conclusion

In this paper we have presented a new method for collecting the variable location information to be incorporated into a variable range table. This is necessary for solving the data location problem, allowing debuggers of optimized code to determine where variable values reside during execution of the program. By taking advantage of key instruction information, our method uses dataflow analysis techniques to collect all the variable location information in a single pass over the final optimized internal representation of the program.

Our approach has many advantages over previous approaches. It is potentially faster than previous approaches, because it collects the data in one dataflow pass, rather than building an initial set of data and evolving the set as optimizations are performed. Unlike previous approaches the person implementing this technique does not need to understand how all of the optimizations in the compiler are implemented. In previous approaches the source code for collecting the variable location data has to be distributed throughout the optimizer code; any time an optimization is added, removed, or modified, the code for collecting the variable location data must be modified as well. By contrast our approach is completely independent of which optimizations the compiler performs and of

⁷ It is possible to construct pathological cases for which this assumption is false; however we do not believe such pathological cases ever actually arise inside compilers, and therefore we feel it is reasonable to make this assumption.

how those optimizations are implemented. The code for performing the dataflow analysis is all in one place, making it easier to write, to maintain, and to debug.

A further benefit of using dataflow analysis on the final representation of the program to collect the variable location data is that this also allows for easy identification of those portions of the target program for which any given variable is either uninitialized or non-resident. Previous variable range tables have not contained this information at all.

Finally by implementing these ideas within an existing commercial compiler and debugger, we have shown that these ideas work and that they can be retrofitted into existing tools without too much effort.

A Algorithm for Generating, Passing, and Killing Location Tuples within a Basic Block

```

currentSet  $\leftarrow \bigcup_{p \in \text{predecessor}(\text{currentBB})} p.\text{locations}$ 
i  $\leftarrow$  first instruction for basic block
do {
  if (i has a destination)
    forall l in currentSet
      if (l.location = i.destination)
        kill(l)
      fi
    endfor
  fi

  if (i is an assignment key instruction)
    varname  $\leftarrow$  name_lookup(i.source_position)
    forall l in currentSet
      if (l.varname = varname)
        kill(l)
      fi
    endfor
    newRecord  $\leftarrow$  gen (varname, i.dest, i.address, undefined)
    currentSet  $\leftarrow$  currentSet  $\cup$  { newRecord }

  else if (i is a register copy)
    forall l in currentSet
      if (l.location = i.source)
        newRecord  $\leftarrow$  gen (l.varname, i.dest, i.address, undefined)
        currentSet  $\leftarrow$  currentSet  $\cup$  { newRecord }
      fi
    endfor

  else if (i is a function call)
    forall l in currentSet
      if (l.location = return value register)
        kill(l)
      fi
    endfor

  else if (i is a memory read)

```



```

if (i.memory_location is "home" address of variable)
  newRecord  $\leftarrow$  gen (name of var, i.dest, i.address, undefined)
  currentSet  $\leftarrow$  currentSet  $\cup$  { newRecord }
else
  forall l in currentSet
    if (l.location = i.mem_loc)
      newRecord  $\leftarrow$  gen (name of var, i.dest, i.address, undefined)
      currentSet  $\leftarrow$  currentSet  $\cup$  { newRecord }
    fi
  endfor
fi
else if (i is a memory write)
  forall l in currentSet
    if (l.location = i.source)
      newRecord  $\leftarrow$  gen(l.varname, i.mem_loc, i.address, undefined)
      currentSet  $\leftarrow$  currentSet  $\cup$  { newRecord }
    fi
  endfor
fi

i  $\leftarrow$  i.next_instruction
} while (i  $\neq$   $\emptyset$ )

```

References

1. A. Adl-Tabatabai and T. Gross, "Evicted Variables and the Interaction of Global Register Allocation and Symbolic Debugging", *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, January 1993, pp. 371-383
2. A. Aho, R. Sethi, and J. Ullman, "Compilers Principles, Techniques, and Tools", Addison-Wesley Publishing Company, 1986
3. D. Coutant, S. Meloy, and M. Ruscetta, "DOC: A Practical Approach to Source-Level Debugging of Globally Optimized Code", In *Proceedings of the 1988 PLDI Conference*, 1988
4. J. Silverstein, ed., "DWARF Debugging Information Format", Proposed Standard, UNIX International Programming Languages Special Interest Group, July 1993
5. C. Tice and S. L. Graham, "Key Instructions: Solving the Code Location Problem for Optimized Code", Research Report 164, Compaq Systems Research Center, Palo Alto, CA, Sept. 2000
6. C. Tice, "Non-Transparent Debugging of Optimized Code", Ph.D. Dissertation, Technical Report UCB//CSD-99-1077, University of California, Berkeley, Oct. 1999.
7. P. Zellweger, "High Level Debugging of Optimized Code", Ph.D. Dissertation, University of California, Berkeley, Xerox PARC TR CSL-84-5, May 1984.

Efficient Symbolic Analysis for Optimizing Compilers^{*}

Robert A. van Engelen

Dept. of Computer Science, Florida State University, Tallahassee, FL 32306-4530
engelen@cs.fsu.edu

Abstract. Because most of the execution time of a program is typically spend in loops, loop optimization is the main target of optimizing and restructuring compilers. An accurate determination of induction variables and dependencies in loops is of paramount importance to many loop optimization and parallelization techniques, such as generalized loop strength reduction, loop parallelization by induction variable substitution, and loop-invariant expression elimination. In this paper we present a new method for induction variable recognition. Existing methods are either ad-hoc and not powerful enough to recognize some types of induction variables, or existing methods are powerful but not safe. The most powerful method known is the symbolic differencing method as demonstrated by the Parafrase-2 compiler on parallelizing the Perfect Benchmarks^(R). However, symbolic differencing is inherently unsafe and a compiler that uses this method may produce incorrectly transformed programs without issuing a warning. In contrast, our method is safe, simpler to implement in a compiler, better adaptable for controlling loop transformations, and recognizes a larger class of induction variables.

1 Introduction

It is well known that the optimization and parallelization of scientific applications by restructuring compilers requires extensive analysis of induction variables and dependencies in loops in order for compilers to effectively transform and optimize loops. Because most of the execution time of an application is spend in loops, restructuring compilers attempt to aggressively optimize loops. To this end, many ad-hoc techniques have been developed for loop induction variable recognition [12,13,20,22] for loop restructuring transformations such as *generalized loop strength reduction*, *loop parallelization by induction variable substitution* (the reverse of strength reduction), and *loop-invariant expression elimination* (code motion). However, these ad-hoc techniques fall short of recognizing *generalized induction variables* (GIVs) with values that form polynomial and geometric progressions through loop iterations [3,7,8,11,19]. The importance of GIV recognition in the parallelization of the Perfect Benchmarks^(R) and other codes was recognized in an empirical study by Singh and Hennessy [14].

^{*} This work was supported in part by NSF grant CCR-9904943

The effectiveness of GIV recognition in the actual parallelization of the Perfect Benchmarks^(R) was demonstrated by the Parafrase-2 compiler [10]. Parafrase-2 uses Haghighat’s symbolic differencing method [10] to detect GIVs. Symbolic differencing is the most powerful induction variable recognition method known.

In this paper we show that symbolic differencing is an unsafe compiler method when not used wisely (i.e. without a user verifying the result). As a consequence, a compiler that adopts this method may produce incorrectly transformed programs without issuing a warning. We present a new induction variable recognition method that is safe and simpler to implement in a compiler and recognizes a larger class of induction variables.

This paper is organized as follows: Section 2 compares our approach to related work. Section 3 presents our generalized induction variable recognition method. Results are given in Section 4. Section 5 summarizes our conclusions.

2 Related Work

Many ad-hoc compiler analysis methods exist that are capable of recognizing *linear induction variables*, see e.g. [12,13,20,22]. Haghighat’s symbolic differencing method [10] recognizes *generalized induction variables* [3,7,8,11,19] that form polynomial and geometric progressions through loop iterations. More formally, a GIV is characterized by its function χ defined by

$$\chi(n) = \varphi(n) + r a^n \quad (1)$$

where n is the loop iteration number, φ is a polynomial of order k , and a and r are loop-invariant expressions.

Parallelization of a loop containing GIVs with (multiple) update assignment statements requires the removal of the updates and the substitution of GIVs in expressions by their closed-form characteristic function χ . This is also known as *induction variable substitution*, which effectively removes all cross-iteration dependencies induced by GIV updates, enabling a loop to be parallelized.

The symbolic differencing method is illustrated in Fig. 1. A compiler symbolically evaluates a loop a fixed number of iterations using *abstract interpretation*. The sequence of symbolic values of each variable are tabulated in *difference tables* from which polynomial and geometric progressions can be recognized. To recognize polynomial GIVs of degree m , a loop is executed at most $m + 2$ iterations. For example, for $m = 3$, the compiler executes the loop shown in Fig. 1(a) five times and constructs a difference table for each variable. The difference table of variable \mathbf{t} is depicted in Fig. 1(b) above the dashed line. According to [10], the compiler can determine that the polynomial degree of \mathbf{t} is $m = 3$, because the final difference is zero (bottom-most zero above the dashed line). Application of induction variable substitution by Parafrase-2 results in the loop Fig. 1(c).

The problem with this approach is that the analysis is incorrect when m is set too low. Six or more iterations are necessary (this includes the diagonal below the dashed line) to find that the degree of \mathbf{t} is actually 5. Hence, the loop shown in Fig. 1(c) is incorrect. Fig. 1(d) depicts the correctly transformed loop.

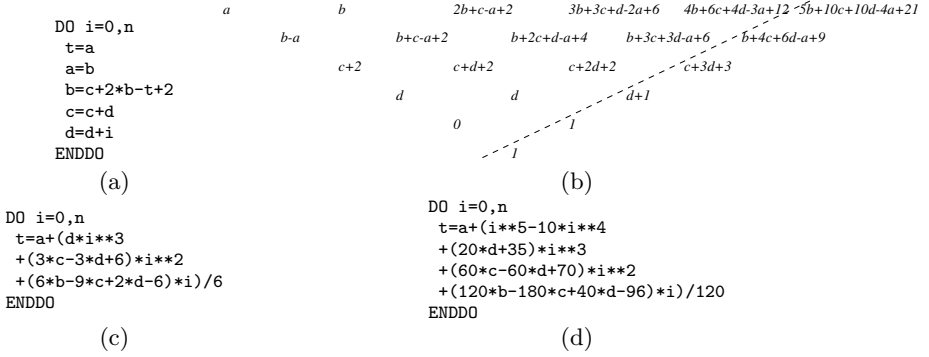


Fig. 1. Example Loop (a), Difference Table of t (b), Incorrectly Transformed Loop (c), and Correctly Transformed Loop (d)

The origin of the problem with symbolic differencing lies in the fact that difference tables form low-order approximations of higher-order polynomials and functions. Hence, symbolic differencing requires a user of the compiler to specify a maximum polynomial order m that is *guaranteed* to be the largest order among all of the known GIVs in a program. This means that the user must be knowledgeable about the type of algorithms in the program to make a wise decision. The optimization of the Perfect Benchmarks^(R) by Parafrase-2 required $m = 3$ to produce correctly transformed programs. Many real-world applications exist that use higher degree polynomials such as in transfinite interpolation for grid generation [12] for CFD and CAD applications, and in hard-coded curve plotting and surface rendering applications, see e.g. [4].

According to [10], symbolic differencing requires extensive symbolic expression manipulation. A symbolic kernel is required with a concise set of algebraic/symbolic operations derived from mathematical tools such as number theory and mathematical induction for performing symbolic operations. The implementation is further hampered by the fact that difference tables can contain large and complicated symbolic entries. In contrast, our method relies on the use of a term rewriting system with 14 rules to derive normal forms for GIVs and 21 rules to derive the closed-form functions from these forms. Our approach requires symbolic manipulation equally powerful as classical constant-folding [1].

In comparing our method to symbolic differencing we find that both methods can handle *multiple assignments to induction variables, generalized induction variables in loops with symbolic bounds and strides, symbolic integer division, conditional induction expressions, cyclic induction dependencies, symbolic forward substitution, symbolic loop-invariant expressions, and wrap-around variables*. However, our method is not capable of detecting *cyclic recurrences*. We found that cyclic recurrences are very rare in the Perfect Benchmarks^(R).

¹ Assuming that symbolic manipulation by constant-folding includes associativity, commutativity, and distributivity of product and addition.

3 Generalized Induction Variable Recognition

In this section we present our induction variable recognition method and associated compiler algorithms. First, we introduce the chains of recurrences formalism that forms the mathematical basis of our approach.

3.1 Chains of Recurrences

Chains of recurrences (CRs) are developed by Bachmann, Zima, and Wang [5] to expedite the evaluation of closed-form functions and expressions on regular grids. The CR algebra enables the construction and simplification of recurrence relations by a computer algebra system. An elementary (scalar) expression can be symbolically transformed into its mathematically equivalent CR [4]. The CR provides a representation that allows it to be translated into a loop construct which efficiently evaluates the expression on a regular grid, similar to loop strength reduction of the original single elementary expression.

Basic Formulation. A closed-form function f evaluated in a loop with loop counter variable i can be rewritten into a mathematical equivalent *System of Recurrence Relations* (SSR) [21] $f_0(i), f_1(i), \dots, f_k(i)$, where the functions $f_j(i)$ for $j = 0, \dots, k-1$ are linear recurrences of the form

$$f_j(i) = \begin{cases} \phi_j & \text{if } i = 0 \\ f_j(i-1) \odot_{j+1} f_{j+1}(i-1) & \text{if } i > 0 \end{cases} \quad (2)$$

with $\odot_{j+1} \in \{+, *\}$, $j = 0, \dots, k-1$, and coefficients ϕ_j are loop-invariant expressions (i.e. induction variables do not occur in ϕ_j). Expression f_k is loop invariant or a similar recurrence system. When the loop is normalized, i.e. $i = 0, \dots, n$ for some $n \geq 0$, it can be shown that $f(i) = f_0(i)$ for all $i = 0, \dots, n$.

A shorthand notation for Eq. (2) is a *Basic Recurrence* (BR) [5]:

$$f_j(i) = \{\phi_j, \odot_{j+1}, f_{j+1}\}_i \quad (3)$$

The BR notation allows the system (2) to be written as

$$\Phi_i = \{\phi_0, \odot_1, \{\phi_1, \odot_2, \dots, \{\phi_{k-1}, \odot_k, f_k\}_i\}_i\}_i \quad (4)$$

When flattened to a single tuple

$$\Phi_i = \{\phi_0, \odot_1, \phi_1, \odot_2, \dots, \odot_k, f_k\}_i \quad (5)$$

it is a *Chain of Recurrences* (CR) [45] with $k = L(\Phi_i)$ the length of the CR.

A CR Φ_i is called *polynomial* if $\odot_j = +$, for all $j = 1, \dots, k$. A polynomial CR has a closed-form function that is a k -order polynomial in variable i . The sequence $\phi_0, \phi_1, \dots, \phi_{k-1}, f_k$ forms the lower-left diagonal of a difference table of the polynomial. CR Φ_i is called *exponential* if $\odot_j = *$, for all $j = 1, \dots, k$. CR $\Phi_i = \{\phi_0, *, \phi_1, +, f_2\}_i$ is called *factorial* if $\phi_1 \geq 1$ and $f_2 = 1$, or $\phi_1 \leq -1$ and $f_2 = -1$.

CR Construction. Fig. 2 depicts \mathcal{CR} rewrite rules adapted from the CR algebra presented in [45]. The proof of correctness of the algebra can be found in [4]. CR construction proceeds by replacing every occurrence of the loop counter variable i (i.e. the basic induction variable) in an expression by the CR $\{a, +, s\}_i$, where a is i 's symbolic initial value and s is the stride. Then, \mathcal{CR} rules are exhaustively applied to the expression [2]. In [16] we proved that \mathcal{CR} is complete (i.e. confluent and terminating) and, hence, CRs are normal forms for polynomials, exponentials, and factorials.

The exhaustive application of \mathcal{CR} results in so-called *CR-expressions*, which are expressions that *contain* CRs as subexpressions [3]. The normalized CR expression of a GIV with characteristic function Eq. (11) is the sum of a polynomial CR and the exponential CR $\{r, *, a\}_n$.

3.2 The Algorithms

We developed a compiler algorithm for induction variable recognition. The algorithm is capable of detecting multiple induction variables in loop hierarchies by exploiting the CR algebra in an entirely novel way. The induction variable recognition method forms the basis of our induction variable substitution algorithm. Induction variable substitution amounts to the removal of induction variable update operations and the replacement of the induction variables by their closed-forms. This requires the inverse rules \mathcal{CR}^{-1} shown in Fig. 3 which we developed to translate CRs back to closed-form functions.

The principle of our algorithm is illustrated in the program fragment below, which demonstrates the key idea to our induction variable recognition method and induction variable substitution:

DO i=0,n ... j=j+h k=k+2*i ENDDO	⇒	DO i=0,n ... j=j+h k=k+{0, +, 2}_i ENDDO	⇒	DO i=0,n ... j={j ₀ , +, h}_i k={k ₀ , +, 0, +, 2}_i ENDDO	⇒	j ₀ =j; k ₀ =k DOALL i=0,n j=j ₀ +h*i k=k ₀ +i*i-i ... ENDDO
--	---	--	---	--	---	---

First, loop counter variable i is replaced by its CR representation $\{0, +, 1\}_i$ upon which rule 2 of \mathcal{CR} Fig. 2 translates $2*i$ into the CR $\{0, +, 2\}_i$. Then, linear induction variable j and non-linear induction variable k are recognized by our algorithm from their update operations and replaced by CRs $\{j_0, +, h\}_i$ and $\{k_0, +, 0, +, 2\}_i$, where j_0 and k_0 are initial values of j and k before the loop. Finally, the \mathcal{CR}^{-1} rules are applied to substitute the CR of j and k with their closed-forms, enabling the loop to be parallelized. The advantage of our approach is that simple algebraic rules are exploited to modify the code from one form into another.

² Applied together with constant folding to simplify CR coefficients.

³ When CRs are mentioned in this text we refer to pure CRs of the form Eq. (5). CR-expressions will be indicated explicitly.

LHS	RHS
1 $E + \{\phi_0, +, f_1\}_i$	$\Rightarrow \{E + \phi_0, +, f_1\}_i$ when E is loop invariant
2 $E * \{\phi_0, +, f_1\}_i$	$\Rightarrow \{E * \phi_0, +, E * f_1\}_i$ when E is loop invariant
3 $E * \{\phi_0, *, f_1\}_i$	$\Rightarrow \{E * \phi_0, *, f_1\}_i$ when E is loop invariant
4 $E^{\{\phi_0, +, f_1\}_i}$	$\Rightarrow \{E^{\phi_0}, *, E^{f_1}\}_i$ when E is loop invariant
5 $\{\phi_0, *, f_1\}_i^E$	$\Rightarrow \{\phi_0^E, *, f_1^E\}_i$ when E is loop invariant
6 $\{\phi_0, +, f_1\}_i + \{\psi_0, +, g_1\}_i$	$\Rightarrow \{\phi_0 + \psi_0, +, f_1 + g_1\}_i$
7 $\{\phi_0, +, f_1\}_i * \{\psi_0, +, g_1\}_i$	$\Rightarrow \{\phi_0 \psi_0, +, \{\phi_0, +, f_1\}_i * g_1 + \{\psi_0, +, g_1\}_i * f_1 + f_1 * g_1\}_i$
8 $\{\phi_0, *, f_1\}_i * \{\psi_0, *, g_1\}_i$	$\Rightarrow \{\phi_0 \psi_0, *, f_1 g_1\}_i$
9 $\{\phi_0, *, f_1\}_i^{\{\psi_0, +, g_1\}_i}$	$\Rightarrow \{\phi_0^{\psi_0}, *, \{\phi_0, *, f_1\}_i^{g_1} * f_1^{\{\psi_0, +, g_1\}_i} * f_1^{g_1}\}_i$
10 $\{\phi_0, +, f_1\}_i!$	$\Rightarrow \begin{cases} \{\phi_0!, *, \left(\prod_{j=1}^{f_1} \{\phi_0 + j, +, f_1\}_i\right)\}_i & \text{if } f_1 \geq 0 \\ \{\phi_0!, *, \left(\prod_{j=1}^{ f_1 } \{\phi_0 + j, +, f_1\}_i\right)^{-1}\}_i & \text{if } f_1 < 0 \end{cases}$
11 $\log\{\phi_0, *, f_1\}_i$	$\Rightarrow \{\log \phi_0, +, \log f_1\}_i$
12 $\{\phi_0, +, 0\}_i$	$\Rightarrow \phi_0$
13 $\{\phi_0, *, 1\}_i$	$\Rightarrow \phi_0$
14 $\{0, *, f_1\}_i$	$\Rightarrow 0$

Fig. 2. \mathcal{CR}

LHS	RHS
1 $\{\phi_0, +, f_1\}_i$	$\Rightarrow \phi_0 + \{0, +, f_1\}_i$ when $\phi_0 \neq 0$
2 $\{\phi_0, *, f_1\}_i$	$\Rightarrow \phi_0 * \{1, *, f_1\}_i$ when $\phi_0 \neq 1$
3 $\{0, +, -f_1\}_i$	$\Rightarrow -\{0, +, f_1\}_i$
4 $\{0, +, f_1 + g_1\}_i$	$\Rightarrow \{0, +, f_1\}_i + \{0, +, g_1\}_i$
5 $\{0, +, f_1 * g_1\}_i$	$\Rightarrow f_1 * \{0, +, g_1\}_i$ when i does not occur in f_1
6 $\{0, +, \log f_1\}_i$	$\Rightarrow \log\{1, *, f_1\}_i$
7 $\{0, +, f_1^i\}_i$	$\Rightarrow \frac{f_1^{i+1} - 1}{f_1 - 1}$ when i does not occur in f_1 and $f_1 \neq 1$
8 $\{0, +, f_1^{g_1 + h_1}\}_i$	$\Rightarrow \{0, +, f_1^{g_1} * f_1^{h_1}\}_i$
9 $\{0, +, f_1^{g_1 * h_1}\}_i$	$\Rightarrow \{0, +, (f_1^{g_1})^{h_1}\}_i$ when i does not occur in f_1 and g_1
10 $\{0, +, f_1\}_i$	$\Rightarrow i * f_1$ when i does not occur in f_1
11 $\{0, +, i\}_i$	$\Rightarrow \frac{i^2 - i}{2}$
12 $\{0, +, i^n\}_i$	$\Rightarrow \sum_{k=0}^n \frac{\binom{n+1}{k}}{n+1} B_k i^{n-k+1}$ for $n \in \mathbb{N}$, B_k is k^{th} Bernoulli number
13 $\{1, *, -f_1\}_i$	$\Rightarrow (-1)^i \{1, *, f_1\}_i$
14 $\{1, *, \frac{1}{f_1}\}_i$	$\Rightarrow \{1, *, f_1\}_i^{-1}$
15 $\{1, *, f_1 * g_1\}_i$	$\Rightarrow \{1, *, f_1\}_i * \{1, *, g_1\}_i$
16 $\{1, *, f_1^{g_1}\}_i$	$\Rightarrow f_1^{\{1, *, g_1\}_i}$ when i does not occur in f_1
17 $\{1, *, g_1^{f_1}\}_i$	$\Rightarrow \{1, *, g_1\}_i^{f_1}$ when i does not occur in f_1
18 $\{1, *, f_1\}_i$	$\Rightarrow f_1^i$ when i does not occur in f_1
19 $\{1, *, i\}_i$	$\Rightarrow 0^i$
20 $\{1, *, i + f_1\}_i$	$\Rightarrow \frac{(i+f_1-1)!}{(f_1-1)!}$ when i does not occur in f_1 and $f_1 \geq 1$
21 $\{1, *, f_1 - i\}_i$	$\Rightarrow (-1)^i * \frac{(i-f_1-1)!}{(-f_1-1)!}$ when i does not occur in f_1 and $f_1 \leq -1$

Fig. 3. \mathcal{CR}^{-1}

IVS(S)

- **input:** statement list S
- **output:** *Induction Variable Substitution* applied to S

CALL *IVStrans*(S)

Use reaching flow information to propagate initial variable values to the CRs in S

Apply CR^{-1} to every CR in S

For every left-over CR Φ_i in S , create an index array ia and code K to initialize the array using algorithm *CRGEN*(Φ_i, b, ia, K), where b is the upper bound of the index array which is the maximum value of the induction variable i of the loop in which Φ_i occurs

IVStrans(S)

- **input:** statement list S
- **output:** *Induction Variable Substitution* applied to S , where CR expressions in S represent the induction expressions of loops in S

FOR each do-loop L in statement list S DO

Let $S(L)$ denote the body statement list of loop L , let I denote the basic induction variable of the loop with initial value expression a , bound expression b , and stride expression s

CALL *IVStrans*($S(L)$)

TRY

CALL *SSA**($S(L), A$)

CALL *CR*($I, a, s, S(L), A$)

CALL *HOIST*($I, a, b, s, S(L), A$)

CATCH FAIL:

Continue with next do-loop L in S

Fig. 4. Algorithm *IVS*

Algorithm *IVS* shown in Fig. 4 applies induction variable substitution to a set of nested loops in a statement list. The programming model supported by *IVS* includes sequences, assignments, if-then-elses, do-loops, and while-loops. *IVS* analyzes a do-loop nest (not necessarily perfectly nested) from the innermost loops, which are the primary candidates for optimization, to the outermost loops. For every loop in the nest, the body is converted to a single-static assignment (SSA) form with assignments to scalar variables separated from the loop body and stored in set A . Next, algorithm *CR* converts the expressions in A to CR-expressions to detect induction variables. Algorithm *HOIST* hoists the induction variable update assignments out of the loop and replaces induction expressions in the loop by closed-forms. If an induction variable has no closed-form, algorithm *CRGEN* is used to create an index array that will serve as a closed-form. This enables *IVS* for a larger class of induction variables compared to GIVs defined by Eq. (1). FAIL indicates failure of the algorithm to apply *IVS*, which can only be due to failure of *SSA** (see *SSA** description later).

The worst-case computational complexity of *IVS* is $\mathcal{O}(kn \log(n) m^2)$, where k is the maximum loop nesting level, n is the length of the source code fragment, and m is the maximum polynomial order of the GIVs in the fragment. This bound is valid provided that Bachmann's CR construction algorithm [4] is used for the fast construction of polynomial CRs.

Note that the *CR* rules in Fig. 2 are applicable to both integer and floating-point typed expressions. Rule 11 handles floating point expressions only. It is guaranteed that integer-valued induction variables have CRs with integer-valued coefficients. However, some integer-valued expressions in a loop that *contain* induction variables may be converted into CRs with rational CR coefficients and a compiler implementation of the rules must handle rationals appropriately.

```

SSA*(S, A)
- input: loop body statement list S
- output: SSA-modified S and partially ordered set A, or FAIL
A := ∅
FOR each statement Si ∈ S from the last (i = |S|) to the first statement (i = 1) DO
  CASE Si
  OF assignment statement of expression X to V:
    IF V is a numeric scalar variable and (V, ⊥) ∉ A THEN
      FOR each statement Sj ∈ S, j = i + 1, ..., |S| DO
        Substitute in Sj every occurrence of variable V by a pointer to X
      FOR each (U, Y) ∈ A DO
        Substitute in Y every occurrence of variable V by a pointer to X
      IF (V, ⊥) ∉ A THEN /* note: _ is a wildcard */
        A := A ∪ {(V, X)}
      Remove Si from S
    ELSE /* assignment to non-numeric or non-scalar variable */
      Continue with next statement Si
  OF if-then-else statement with condition C, then-clause S(T), and else-clause S(E):
    CALL SSA*(S(T), A1)
    CALL SSA*(S(E), A2)
    CALL MERGE(C, A1, A2, A1,2)
    FOR each (V, X) ∈ A1,2 DO
      FOR each statement Sj, j = i + 1, ..., |S| DO
        Substitute in Sj every occurrence of variable V by a pointer to X
      FOR each (U, Y) ∈ A DO
        Substitute in Y every occurrence of V by a pointer to X
      IF (V, ⊥) ∉ A THEN /* note: _ is a wildcard */
        A := A ∪ {(V, X)}
  OF do-loop OR while-loop:
    IF the loop body contains an assignment to a scalar numeric variable V THEN
      A := A ∪ (V, ⊥)
  Topologically sort A with respect to <, FAIL if sort not possible (i.e. < is not a partial order on A)
MERGE(C, A1, A2, A1,2)
- input: Boolean expression C, variable-expression sets A1 and A2
- output: merged set A1,2
A1,2 := ∅
FOR each (V, X) ∈ A1 DO
  IF (V, Y) ∈ A2 for some expression Y THEN
    A1,2 := A1,2 ∪ {(V, C?X:Y)}
  ELSE
    A1,2 := A1,2 ∪ {(V, C?X:V)}
FOR each (V, X) ∈ A2 DO
  IF (V, ⊥) ∉ A1,2 THEN
    A1,2 := A1,2 ∪ {(V, C?V:X)}

```

Fig. 5. Algorithm *SSA**

Single Static Assignment Form. The *SSA** algorithm shown in Fig. 5 uses the precedence relation on variable-expression pairs defined by

$$(U, Y) \prec (V, X) \quad \text{if } U \neq V \text{ and } V \text{ occurs in } Y$$

to obtain an ordered set of variable-expression pairs *A* extracted from a loop body. The algorithm constructs a directed acyclic graph for the expressions in *A* and the expressions in the loop body such that common-subexpressions share the same node in the graph to save space and time. Thus, when a subexpression is transformed by rules \mathcal{CR} or \mathcal{CR}^{-1} the result is immediately visible to the expressions that refer to this subexpression. This works best when the rules \mathcal{CR} and \mathcal{CR}^{-1} are applied to an expression from the innermost to the outermost redexes (i.e. normal-order reduction).

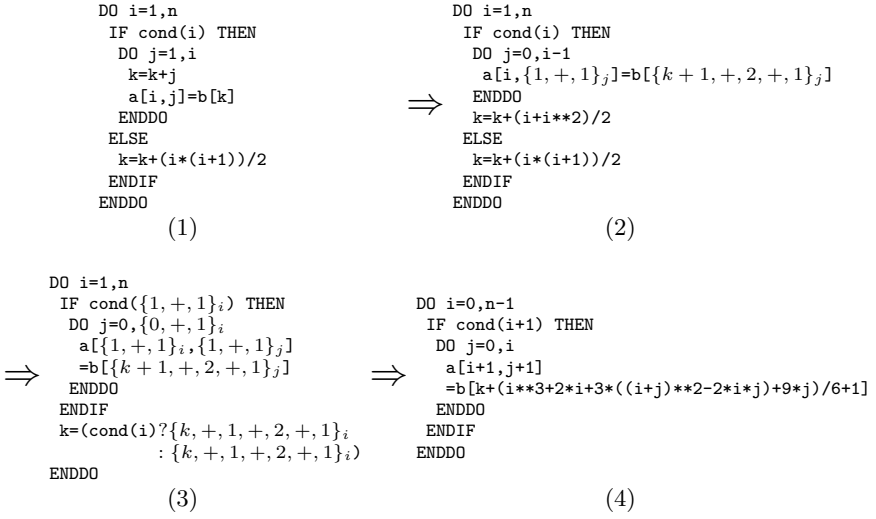


Fig. 6. Example Analysis of Conditional Induction Variables

Each variable has only one expression in A . Set A contains the potential induction variables of the loop. Values of conditional induction variables are represented by conditional expressions of the form $C?X:Y$, where C is the condition and X and Y are expressions. The conditional expression $C?X:X$ is rewritten⁴ into X . An example conditional induction variable analysis is shown in Fig. 6. For sake of simplicity, we assume that the values of conditions do not change when the conditions are replicated to different parts of the loop body. The algorithm can be easily modified to produce code in which a temporary variable is used to hold the value of a condition when conditions cannot be replicated.

Algorithm SSA^* fails when \prec is not a partial order on A . The source of this problem is the presence of cyclic recurrences in a loop, see e.g. [10] (p. 29). Cyclic dependencies are not a problem.

Induction Variable Recognition and Substitution. Algorithm CR shown in Fig. 7 converts the expressions in A into normalized CR-expressions to detect induction variables.

Fig. 8 illustrates the analysis of *wrap-around variables* by algorithm CR . The code example is from [10] (p. 52). A wrap-around variable has at least one use of the variable before its (first) assignment statement in the loop body, as is the case for variables j and k in Fig. 8(1). For wrap-around variable analysis, each use of the variable V before its definition is replaced by the CR-expression

$$\{V - \mathcal{V}(\mathcal{B}(\Phi_i)), *, 0\}_i + \mathcal{B}(\Phi_i)$$

⁴ Recall that CRs are normal forms, and when X is a CR this rewrite rule is trivial to implement.

$CR(I, a, s, S, A)$
- **input:** loop induction variable I with initial value a and stride s ,
statement list S , and topologically ordered set S of variable-expression pairs
- **output:** expressions in A are converted to CR-expressions
FOR each $(V, X) \in A$ in topological order (\prec) DO
Substitute in X every occurrence of I by $\{a, +, s\}_I$
Apply \mathcal{CR} rules to X
IF X is of the form $V + C$, where C is a loop invariant expression or a CR THEN
Replace (V, X) in A with $(V, \{V, +, C\}_I)$
FOR each $(U, Y) \in A$, $(V, X) \prec (U, Y)$ DO
Substitute every occurrence of V in Y by $\{V, +, C\}_I$
ELSE IF X is of the form $V * C$, where C is a loop invariant expression or a CR THEN
Replace (V, X) in A with $(V, \{V, *, C\}_I)$
FOR each $(U, Y) \in A$, $(V, X) \prec (U, Y)$ DO
Substitute every occurrence of V in Y by $\{V, *, C\}_I$
ELSE IF V does not occur in X THEN /* **wrap-around variable** */
Replace (V, X) in A with $(V, \{V - \mathcal{V}(\mathcal{B}(X)), *, 0\}_I + \mathcal{B}(X))$
FOR each $(U, Y) \in A$, $(V, X) \prec (U, Y)$ DO
Substitute every occurrence of V in Y by $\{V - \mathcal{V}(\mathcal{B}(X)), *, 0\}_I + \mathcal{B}(X)$
ELSE /* **other type of assignment** */
Continue with next (V, X) pair in A
FOR each $(V, X) \in A$ in topological order (\prec) DO
FOR each statement $S_i \in S$ (and statements at deeper nesting levels) DO
Substitute every occurrence of V in S_i by X
Apply \mathcal{CR} rules to every expression in S_i

Fig. 7. Algorithm CR

<pre> j=m DO i=m,n a[j]=b[k] j=i+1 k=j ENDDO </pre> <p>(1)</p>	\Rightarrow	<pre> j=m DO i=m,n a[j]=b[k] j={j-m,*,0}_i +{m,+,1}_i k={k-m,*,0}_i +{m,+,1}_i ENDDO </pre> <p>(2)</p>	\Rightarrow	<pre> j=m DO i=0,n-m a[{j-m,*,0}_i +{m,+,1}_i] =b[{k-m,*,0}_i +{m,+,1}_i] ENDDO </pre> <p>(3)</p>	\Rightarrow	<pre> DO i=0,n-m a[i+m]=b[0**i*(k-m)+i+m] ENDDO </pre> <p>(4)</p>
--	---------------	--	---------------	---	---------------	---

Fig. 8. Example Analysis of Wraparound Variables

where Φ_i is the CR-expression of the variable V . The symbolic functions \mathcal{V} and \mathcal{B} are described in [16] with their proof of correctness. The closed-form of the CR $\{\phi_0, *, 0\}_i$ is $\phi_0 * 0^i$, which evaluates in Fortran to ϕ_0 if $i = 0$ and 0 if $i \neq 0$.

Algorithm *HOIST* shown in Fig. 9 applies the final stage by hoisting induction variable update assignments out of the loop. It is assumed that $a \leq b - 1$ for lower bound a and upper bound b of each loop counter variable with positive stride $s > 0$, and $a \geq b - 1$ for $s < 0$. When this constraint is not met for a particular loop, the same approach as in [10] (p. 82) is used in which the loop counter variable i is replaced with $\max(\lfloor (b - a + s)/s \rfloor, 0)$ in the closed-form of an induction variable to set the final value of the variable at the end of the loop.

Induction Variables and Loop Strength Reduction. Polynomial, factorial, GIV, and exponential CR-expressions can always be converted to a closed-form. However, some types of CRs do not have equivalent closed-forms. To extend our approach beyond traditional GIV recognition, we use algorithm *CRGEN*

HOIST(I, a, b, s, S, A)

- **input:** loop induction variable I with initial value a , bound b , stride s ,
 loop statement list S , and A the set of variable-expression pairs

- **output:** loop S with induction variable assignments hoisted out of S

$T := \emptyset$

FOR each $(V, X) \in A$ in reversed topological order (\prec) DO

 Apply \mathcal{CR}^{-1} to X resulting in Y

 IF V does not occur in Y THEN

 IF V is live at the end of the loop THEN

 Substitute every occurrence of I in Y by $\lfloor \frac{b-a+s}{s} \rfloor$

 Append $V := Y$ at the end of T

 ELSE

 Append $V := X$ at the end of S

Replace the statement list S with a loop with body S and followed by statements T :

$S := (\text{DO } I=0, \lfloor \frac{b-a}{s} \rfloor \text{ } S \text{ ENDDO } T)$

Fig. 9. Algorithm *HOIST*

CRGEN(Φ_i, b, id, S)

- **input:** CR $\Phi_i = \{\phi_0, \odot_1, \dots, \odot_k, f_k\}_i$, bound expression $b \geq 0$, and identifier id
 - **output:** statement list S to numerically evaluate Φ_i storing the values in $id[0..b]$

S is the statement list created from the template below, where $cr_j, j = 1, \dots, k$,
 are temporary scalar variables of type integer if all ϕ_j are integer, float otherwise:

$id[0] = \phi_0$

$cr_1 = \phi_1$

 :

$cr_k = f_k$

DO $i = 0, b - 1$

$id[i + 1] = id[i] \odot_1 cr_1$

$cr_1 = cr_1 \odot_2 cr_2$

 :

$cr_{k-1} = cr_{k-1} \odot_k cr_k$

ENDDO

Fig. 10. Algorithm *CRGEN*

adopted from algorithm CREval [4] (see [4] for the proof of correctness of the algorithm). Algorithm *CRGEN* shown in Fig. 10 stores CR values in an array. *CRGEN* can be used to evaluate CRs that have or do not have closed-forms.

Algorithm *CRGEN* can be used for *generalized loop strength reduction* as well, to replace GIVs by recurrences that are formed by iterative updates to induction variables. For example, the CR of the closed-form expression $(i*i-i)/2$ is $\{0, +, 0, +, 1\}_i$, assuming that loop index variable i starts with zero and has stride one. The program fragment S produced by *CRGEN*($\{0, +, 0, +, 1\}_i, n, k, S$) calculates $(i*i-i)/2$ with a strength reduced loop (in a slightly different form):

<pre> k=0 cr1=0 DO i=0,n-1 k=k+cr1 cr1=cr1+1 ENDDO </pre>	which is equivalent to	<pre> k=0 DO i=0,n-1 /* k≡(i*i-i)/2 */ k=k+i ENDDO </pre>
---	------------------------	---

This approach has an important advantage compared to symbolic differencing for generalized loop strength reduction. It automatically handles cases in which a loop body contains non-differentiable operators with arguments that are induction expressions. For example, suppose that $\text{MAX}((i*i-i)/2, 0)$ occurs in a

loop body and assume that i is some induction variable. Our method recognizes $(i*i-i)/2$ automatically as an induction expression which, for example, can be loop strength reduced by replacing $(i*i-i)/2$ with a new induction variable. The symbolic differencing method is expensive to use for generalized loop strength reduction to detect subexpressions that are optimizable, because symbolic difference tables have to be constructed for *each* subexpression.

Interprocedural Analysis. Due to limitations in space, the presentation of the *IVS* algorithm in this paper does not include interprocedural analysis. Subroutine inlining can be used but this may result in an explosion in code size. Instead, interprocedural analysis can be performed with algorithm *IVS* by treating the subroutine call as a jump to the routine’s code and back, and by doing some work at the subroutine boundaries to ensure a proper passing of induction variables.

4 Results

In this section we give the results of applying the *IVS* algorithm to code segments of MDG and TRFD. We created a prototype implementation of *IVS* in CTADEL [17,18] to translate MDG, TRFD, and the other code fragments shown in this paper. The MDG code segment is part of a predictor-corrector method in an N-body molecular dynamics application. The TRFD program has been extensively analyzed in the literature, see e.g. [9,10], because its main loop is hard to parallelize due to the presence of a number of coupled non-linear induction variables. Performance results on the parallelization of MDG and TRFD with *IVS* are presented in [10]. We will not present an empirical performance comparison in this paper. Instead, we demonstrate a step-by-step application of the GIV recognition method and *IVS* algorithm on MDG and TRFD.

MDG. Fig. [11](2) depicts the recognition of the ikl and ji GIV updates represented by CRs. Fig. [11](3) shows them hoisted out of the inner loop. This is followed by an analysis of the middle loop which involves forward substitution of the assignments $ji=jiz$ and $ikl=ik+m$. Again, ikl is recognized as an induction variable of the middle loop together with ik , while ji is loop invariant with respect to k . Hoisting of these variable assignments results in Fig. [11](5). After substitution of the ik and jiz induction variables in Fig. [11](5), the *IVS* translated code is shown in Fig. [11](6). Note that arrays c and v in the inner loop in Fig. [11](5) are indexed by *nested* CRs. This form is automatically obtained, in which the CR coefficients of an outer CR may consist of CRs that are guaranteed to be loop invariant with respect to the outer CR’s index variable.

TRFD. The application of *IVS* on TRFD is shown in Fig. [12]. The inner loop is analyzed first from which GIVs are eliminated resulting in Fig. [12](2). Working further outwards, subsequent GIVs are eliminated resulting in Fig. [12](3) and (4). The fully *IVS* transformed TRFD fragment is shown in Fig. [12](5). The complete loop nest can be parallelized on e.g. a shared-memory multiprocessor machine.

```

ik=1
jiz=2
DO i=1,n
  DO k=1,m
    ji=jiz
    ikl=ik+m
    s=0.0
    DO l=1,n
      s=s+c[ji]*v[ikl]
      ikl=ikl+m
      ji=ji+1
    ENDDO
    v[ik]=v[ik]+s
    ik=ik+1
  ENDDO
  jiz=jiz+n+1
ENDDO

```

(1)

```

ik=1
jiz=2
DO i=1,n
  DO k=1,m
    ji=jiz
    ikl=ik+m
    s=0.0
    DO l=1,n
      s=s+c[{ji,+,1}l]
      *v[{ikl,+,m}l]
      ikl={ikl,+,m}l
      ji={ji,+,1}l
    ENDDO
    v[ik]=v[ik]+s
    ik=ik+1
  ENDDO
  jiz=jiz+n+1
ENDDO

```

(2)

```

ik=1
jiz=2
DO i=1,n
  DO k=1,m
    ji=jiz
    ikl=ik+m
    s=0.0
    DO l=0,n-i
      s=s+c[{ji,+,1}l]
      *v[{ikl,+,m}l]
    ENDDO
    ikl=ikl+(n-i+1)*m
    ji=ji+n-i+1
    v[ik]=v[ik]+s
    ik=ik+1
  ENDDO
  jiz=jiz+n+1
ENDDO

```

(3)

```

ik=1
jiz=2
DO i=1,n
  DO k=1,m
    s=0.0
    DO l=0,n-i
      s=s+c[{jiz,+,1}l]
      *v[{ik+m,+,1}k]
      ,+,m}l]
    ENDDO
    v[{ik,+,1}k]
    =v[{ik,+,1}k]+s
    ikl={ik+m+m}
    *(n-i+1,+,1}k
    ji=jiz+n-i+1
    ik={ik,+,1}k
  ENDDO
  jiz=jiz+n+1
ENDDO

```

(4)

```

ik=1
jiz=2
DO i=1,n
  DO k=0,m-1
    s=0.0
    DO l=0,n-{1,+,1}i
      s=s+c[{jiz,+,n+1}i,+,1}l]
      *v[{ik+m,+,m}i,+,1}k]
      ,+,m}l]
    ENDDO
    v[{ik,+,m}i,+,1}k]
    =v[{ik,+,m}i,+,1}k]+s
  ENDDO
  ik={ik,+,m}i
  jiz={jiz,+,n+1}i
ENDDO

```

(5)

```

DO i=0,n-1
  DO k=0,m-1
    s=0.0
    DO j=0,n-i-1
      s=s+c[1+i*(n+1)+2]
      *v[k+m*(i+1+1)+1]
    ENDDO
    v[k+i*m+1]=v[k+i*m+1]+s
  ENDDO
ENDDO

```

(6)

Fig. 11. Algorithm *IVS* Applied to Code Segment of MDG

5 Conclusions

In this paper we presented a novel approach to generalized induction variable recognition for optimizing compilers. We have shown that the method can be used for generalized induction variable substitution, generalized loop strength reduction, and loop-invariant expression elimination. In contrast to the symbolic differencing method, our method is safe and can be implemented with as little effort as adding a compiler phase that includes rewrite rules for chains of recurrences (CRs). In our approach, CR-normalized representations are obtained for a larger class of induction variables than GIVs (e.g. factorials and exponentials).

We are currently investigating the application of the CR method in the areas of program equivalence determination, program correctness proofs, and in optimizing compiler validation techniques [15] to deal with loop semantics. The induction variable recognition method described in this paper can also be used for non-linear dependence testing, see [16]. The basic idea is that dependence

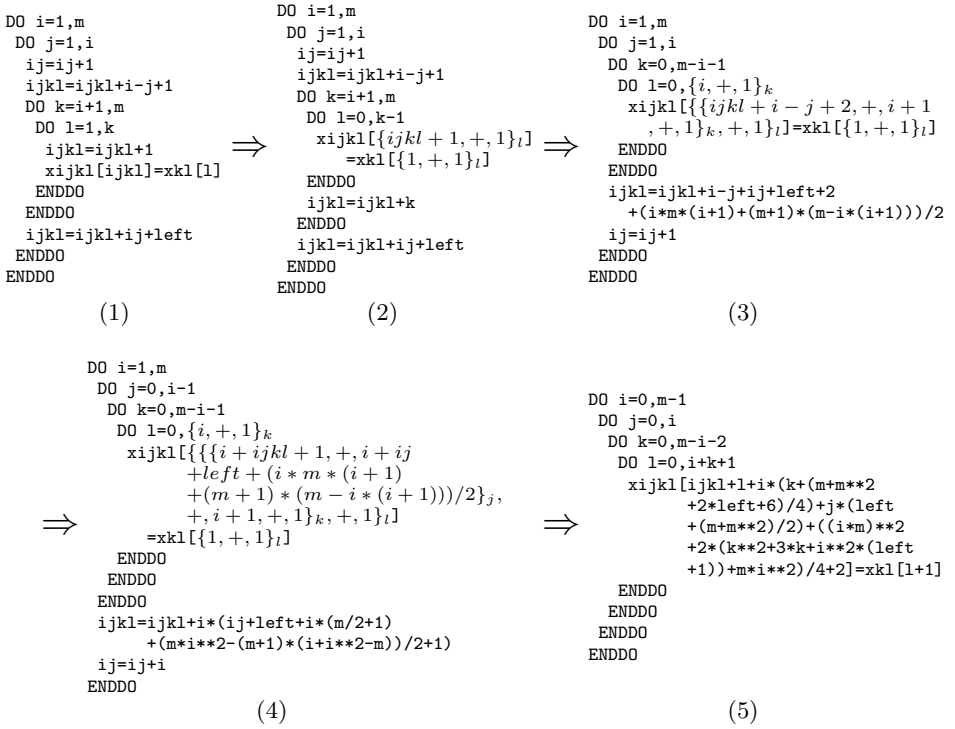


Fig. 12. Algorithm IVS Applied to Code Segment of TRFD

distance vectors are obtained by subtracting two CR-normalized array index expressions, normalizing the result with the \mathcal{CR} rules, and testing for the resulting sign of the CR. No other existing non-linear dependence testing method is as fast as this approach. Other existing compiler techniques for dependence testing are based on value range analysis, first introduced in [6]. Fahringer [9] improved these methods for non-linear dependence testing. The method is particularly suitable for analysis of dependencies across multiple loop levels. These methods are complementary and can be combined with our approach.

References

1. A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley Publishing Company, Reading MA, 1985.
2. F. Allen, J. Cocke, and K. Kennedy. Reduction of operator strength. In S. Munchnick and N. Jones, editors, *Program Flow Analysis*, pages 79–101, New-Jersey, 1981. Prentice-Hall.
3. Z. Ammerguallat and W.L. Harrison III. Automatic recognition of induction variables and recurrence relations by abstract interpretation. In *ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 283–295, White Plains, NY, 1990.

4. O. Bachmann. *Chains of Recurrences*. PhD thesis, Kent State University of Arts and Sciences, 1996.
5. O. Bachmann, P.S. Wang, and E.V. Zima. Chains of recurrences - a method to expedite the evaluation of closed-form functions. In *International Symposium on Symbolic and Algebraic Computing*, pages 242–249, Oxford, 1994. ACM.
6. W. Blume and R. Eigenmann. Demand-driven, symbolic range propagation. In *8th International workshop on Languages and Compilers for Parallel Computing*, pages 141–160, Columbus, Ohio, USA, August 1995.
7. R. Eigenmann, J. Hoeflinger, G. Jaxon, Z. Li, and D.A. Padua. Restructuring fortran programs for cedar. In *ICPP*, volume 1, pages 57–66, St. Charles, Illinois, 1991.
8. R. Eigenmann, J. Hoeflinger, Z. Li, and D.A. Padua. Experience in the automatic parallelization of four perfect-benchmark programs. In *4th Annual Workshop on Languages and Compilers for Parallel Computing, LNCS 589*, pages 65–83, Santa Clara, CA, 1991. Springer Verlag.
9. T. Fahringer. Efficient symbolic analysis for parallelizing compilers and performance estimators. *Supercomputing*, 12(3):227–252, May 1998.
10. Mohammad R. Haghighat. *Symbolic Analysis for Parallelizing Compilers*. Kluwer Academic Publishers, 1995.
11. M.R. Haghighat and C.D. Polychronopoulos. Symbolic program analysis and optimization for parallelizing compilers. In *5th Annual Workshop on Languages and Compilers for Parallel Computing, LNCS 757*, pages 538–562, New Haven, Connecticut, 1992. Springer Verlag.
12. P. Knupp and S. Steinberg. *Fundamentals of Grid Generation*. CRC Press, 1994.
13. S. Munchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Fransisco, CA, 1997.
14. J.P. Singh and J.L. Hennessy. An emperical investigation of the effectiviness and limitations of automatic parallelization. In N. Suzuki, editor, *Shared Memory Multiprocessing*, pages 203–207. MIT press, Cambridge MA, 1992.
15. R. van Engelen, D. Whalley, and X. Yuan. Automatic validation of code-improving transformations. In *ACM SIGPLAN Workshop on Language, Compilers, and Tools for Embedded Systems*, 2000.
16. R.A. van Engelen. Symbolic evaluation of chains of recurrences for loop optimization. Technical report, TR-000102, Computer Science Department, Florida State University, 2000. Available from <http://www.cs.fsu.edu/~engelen/cr.ps.gz>.
17. R.A. van Engelen, L. Wolters, and G. Cats. CTADEL: A generator of multi-platform high performance codes for pde-based scientific applications. In *10th ACM International Conference on Supercomputing*, pages 86–93, New York, 1996. ACM Press.
18. R.A. van Engelen, L. Wolters, and G. Cats. Tomorrow's weather forecast: Automatic code generation for atmospheric modeling. *IEEE Computational Science & Engineering*, 4(3):22–31, July/September 1997.
19. M.J. Wolfe. Beyond induction variables. In *ACM SIGPLAN'92 Conference on Programming Language Design and Implementation*, pages 162–174, San Fransisco, CA, 1992.
20. M.J. Wolfe. *High Performance Compilers for Parallel Computers*. Addison-Wesley, Redwood City, CA, 1996.
21. E.V. Zima. Recurrent relations and speed-up of computations using computer algebra systems. In *DISCO'92*, pages 152–161. LNCS 721, 1992.
22. H. Zima. *Supercompilers for Parallel and Vector Computers*. ACM Press, New York, 1990.

Interprocedural Shape Analysis for Recursive Programs

Noam Rinetzky^{*1} and Mooly Sagiv²

¹ Computer Science Department, Technion, Technion City, Haifa 32000, Israel
maon@cs.technion.ac.il

² Computer Sciences Department, Tel-Aviv University, Tel-Aviv 69978, Israel
msagiv@acm.org

Abstract. A shape-analysis algorithm statically analyzes a program to determine information about the heap-allocated data structures that the program manipulates. The results can be used to optimize, understand, debug, or verify programs. Existing algorithms are quite imprecise in the presence of recursive procedure calls. This is unfortunate, since recursion provides a natural way to manipulate linked data structures. We present a novel technique for shape analysis of recursive programs. An algorithm based on our technique has been implemented. It handles programs manipulating linked lists written in a subset of C. The algorithm is significantly more precise than existing algorithms. For example, it can verify the absence of memory leaks in many recursive programs; this is beyond the capabilities of existing algorithms.

1 Introduction

A shape-analysis algorithm statically analyzes a program to determine information about the heap-allocated data structures that the program manipulates. The analysis algorithm is *conservative*, i.e., the discovered information is true for every input. The information can be used to understand, verify, optimize [6], or parallelize [18,12] programs. For example, it can be utilized to check at compile-time for the absence of certain types of memory management errors, such as memory leakage or dereference of null pointers [5].

This paper addresses the problem of shape analysis in the presence of recursive procedures. This problem is important since recursion provides a natural way to manipulate linked data structures. We present a novel interprocedural shape analysis algorithm for programs manipulating linked lists. Our algorithm analyzes recursive procedures more precisely than existing algorithms. For example, it is able to verify that all the recursive list-manipulating procedures of a small library we experimented with always return a list and never create memory leaks (see Sect. 5). In fact, not only can our algorithm verify that correct programs do not produce errors, it can also find interesting bugs in incorrect programs. For instance, it correctly finds that the recursive procedure `rev` shown

^{*} Partially supported by the Technion and the Israeli Academy of Science.

<pre>typedef struct node{ int d; struct node *n; } *L;</pre>	<pre>L rev(L x) { l₀: L xn, t; if (x == NULL) return NULL; xn = x->n; x->n = NULL; l₁: t = rev(xn); return app(t, x); l₂: }</pre>	<pre>void main() { L hd, z ; hd = create(8); l₃: z = rev(hd); }</pre>
(a)	(b)	(c)
<pre>L create(int s) { L tmp, t1; if (s <= 0) return NULL; t1 = create(s-1); tmp = (L) malloc(sizeof(*L)); tmp->n = t1; tmp->d = s; return tmp; }</pre>		<pre>L app(L p, L q) { L r; if (p == NULL) return q; r = p; while(r->n != NULL) r = r->n; r->n = q; return p; }</pre>
(d)		(e)

Fig. 1. (a) A type declaration for singly linked lists. (b) A recursive procedure which reverses a list in two stages: reverse the tail of the original list and store the result in `t`; then append the original first element at the end of the list pointed to by `t`. (c) The `main` procedure creates a list and then reverses it. We also analyzed this procedure with a recursive version of `app` (see Sect. 5.) (d) A recursive procedure that creates a list. (e) A non-recursive procedure that appends the list pointed to by `q` to the end of the list pointed to by `p`

in Fig. 1(b), which reverses a list (declared in Fig. 1(a)) returns an acyclic linked list and does not create memory leaks. Furthermore, if an error is introduced by removing the statement `x->n = NULL`, the resultant program creates a cyclic list, which leads to an infinite loop on some inputs. Interestingly, our analysis locates this error. Such a precise analysis of the procedure `rev` is quite a difficult task since (i) `rev` is recursive, and thus there is no bound on the number of activation records that can be created when it executes; (ii) the global store is updated destructively in each invocation; and (iii) the procedure is not tail recursive: It sets the value of the local variable `x` before the recursive call and uses it as an argument to `app` after the call ends. No other shape-analysis algorithm we know of is capable of producing results with such a high level of precision for programs that invoke this, or similar, procedures.

A shape-analysis algorithm, like any other static program-analysis algorithm, is forced to represent execution states of potentially unbounded size in a bounded way. This process, often called *summarization*, naturally entails a loss of information. In the case of interprocedural analyses, it is also necessary to summarize all incarnations of recursive procedures in a bounded way.

Shape-analysis algorithms can analyze linked lists in a fairly precise way, e.g., see [15]. For an interprocedural analysis, we therefore follow the approach suggested in [4,11] by summarizing activation records in essentially the same way linked list elements are summarized. By itself, this technique does not retain the precision we would like. The problem is with the (abstract) values obtained for local variables after a call. The abstract execution of a procedure call forces the analysis to summarize, and the execution of the corresponding return has the problem of recovering the information lost at the call. Due to the lack of enough information about the potential values of the local variables, the analysis must make overly conservative assumptions. For example, in the `rev` procedure, if the analysis is not aware of the fact that each list element is pointed to by no more than one instance of the variable `x`, it may fail to verify that `rev` returns an acyclic list (see Example 4.3).

An important concept in our algorithm is the identification of certain global properties of the heap elements pointed to by a local (stack-allocated) pointer variable. These properties describe potential and definite aliases between pointer access paths. This allows the analysis to handle return statements rather precisely. For example, in the `rev` procedure shown in Fig. 1(b), the analysis determines that the list element pointed to by `x` is different from all the list elements reachable from `t` just before the `app` procedure is invoked, which can be used to conclude that `app` must return an acyclic linked list. Proving that no memory leaks occur is achieved by determining that if an element of the list being reversed is not reachable from `t` at l_1 , then it is pointed to by at least one instance of `x`.

A question that comes to mind is how our analysis determines such global properties in the absence of a specification. Fortunately, we found that a small set of “local” properties of the stack variables in the analyzed program can be used to determine many global properties. Furthermore, our analysis does not assume that a local property holds for the analyzed program. Instead, the analysis determines the stack variables that have a given property. Of course, it can benefit from the presence of a specification, e.g., [9], which would allow us to look for the special global properties of the specified program.

For example, the property $sh_x(v)$ holds for a list element v that is pointed to by two or more invisible instances of the parameter `x` from previous activation records. When $sh_x(v)$ does not hold for any list element, we have a guarantee that no list element is pointed to by more than one instance of the variable `x`. This simple local property plays a vital rule in verifying that the procedure `rev` returns an acyclic list (see Example 4.3). Interestingly, this property also sheds some light on the importance of tracking the sharing properties of stack variables. Existing intraprocedural shape-analysis algorithms [2,10,14,15] only record sharing properties of the heap since the number of variables is fixed in the intraprocedural setting. However, in the presence of recursive calls, different incarnations of a local variable may point to the same heap cell.

The ability to have distinctions between invisible instances of variables based on their local properties is the reason for the difference in precision between our method and the methods described in [12,7,8,12,14]. In Sect. 4, we also exploit

properties that capture relationships between the stack and the heap. In many cases, the ability to have these distinctions also leads to a more efficient analysis. Technically, these properties and the analysis algorithm itself are explained (and implemented) using the 3-valued logic framework developed in [13,15]. While our algorithm can be presented in an independent way, this framework provides a sound theoretical foundation for our ideas and immediately leads to the prototype implementation described in Sect. 5. Therefore, Sect. 3 presents a basic introduction to the use of 3-valued logic for program analysis.

2 Calling Conventions

In this section, we define our assumption about the programming language calling conventions. These conventions are somewhat arbitrary; in principle, different ones could be used with little effect on the capabilities of the program analyzer. Our analysis is not effected by the value of non-pointer variables. Thus, we do not represent scalars (conservatively assuming that any value is possible, if necessary), and in the sequel, restrict our attention to pointer variables.

Without loss of generality, we assume that all local variables have unique names. Every invoked procedure has an activation record in which its local variables and parameters are stored. An invocation of procedure f at a *call-site label* is performed in several steps: (i) store the values of actual parameters and *label* in some designated global variables; (ii) at the *entry-point* of f , create a new activation record at the top of the stack and copy values of parameters and *label* into that record; (iii) execute the statements in f until a **return** statement occurs or f 's *exit-point* is reached (we assume that a return statement stores the return value in a designated global variable and transfers the control to f 's exit-point); (iv) at f 's exit-point, pop the stack and transfer control back to the matching *return-site* of *label*; (v) at the return-site, copy the return value if needed and resume execution in the caller.

The activation record at the top of the stack is referred to as the *current activation record*. Local variables and parameters stored in the current activation record and global variables are called *visible*; local variables and parameters stored in other activation records are *invisible*.

2.1 The Running Example

The C program whose **main** procedure shown in Fig. 1(c) invokes **rev** on a list with eight elements. This program is used throughout the paper as a running example. In procedure **rev**, label l_1 plays the role of the recursive call site, l_0 that of **rev**'s entry point, and l_2 of **rev**'s exit point.

3 The Use of 3-Valued Logic for Program Analysis

The algorithm is explained (and implemented) using the 3-valued logic framework developed in [13,15]. In this section, we summarize that framework, which shows how 3-valued logic can serve as the basis for program analysis.

Table 1. The core predicates used in this paper. There is a separate predicate g for every global program variable \mathbf{g} , x for every local variable or parameter \mathbf{x} , and cs_{label} for every label $label$ immediately preceding a procedure call

Predicate	Intended Meaning
$heap(v)$	v is a heap element.
$stack(v)$	v is an activation record.
$cs_{label}(v)$	$label$ is the call-site of the procedure whose activation record is v .
$g(v)$	The heap element v is pointed to by a global variable \mathbf{g} .
$n(v_1, v_2)$	The \mathbf{n} -component of list element v_1 points to the list element v_2 .
$top(v)$	v is the current activation record.
$pr(v_1, v_2)$	The activation record v_2 is the immediate previous activation record of v_1 in the stack.
$x(v_1, v_2)$	The local (parameter) variable \mathbf{x} , which is stored in activation record v_1 , points to the list element v_2 .

3.1 Representing Memory States via 2-Valued Logical Structures

A *2-valued logical structure* S is comprised of a set of individuals (nodes) called a universe, denoted by U^S , and an interpretation over that universe for a set of predicate symbols called the *core predicates*. The interpretation of a predicate symbol p in S is denoted by p^S . For every predicate p of arity k , p^S is a function $p^S: (U^S)^k \rightarrow \{0, 1\}$.

In this paper, 2-valued logical structures represent memory states. An individual corresponds to a memory element: either a heap cell (a list element) or an activation record. The core predicates describe atomic properties of the program memory state. The properties of each memory element are described by unary core predicates. The relations that hold between two memory elements are described by binary core predicates. The core predicates' intended meaning is given in Table 1. This representation intentionally ignores the specific values of pointer variables (i.e., the specific memory addresses that they contain), and record only certain relationships that hold among the variables and memory elements:

- Every individual v represents either a heap cell in which case $heap^S(v) = 1$, or an activation record, in which case $stack^S(v) = 1$.
- The unary predicate cs_{label} indicates the call-site at which a procedure is invoked. Its similarities with the call-strings of [16] are discussed in Sect. 6.
- The unary predicate top is true for the current activation record.
- The binary relation n captures the \mathbf{n} -successor relation between list elements.
- The binary relation pr connects an activation record to the activation record of the caller.
- For a local variable or parameter named \mathbf{x} , the binary relation x captures its value in a specific activation record.

2-valued logical structures are depicted as directed graphs. A directed edge between nodes u_1 and u_2 that is labeled with binary predicate symbol p indicates that $p^S(u_1, u_2) = 1$. Also, for a unary predicate symbol p , we draw p inside a node u when $p^S(u) = 1$; conversely, when $p^S(u) = 0$ we do not draw p in u . For clarity, we treat the unary predicates *heap* and *stack* in a special way; we draw nodes u having $\text{heap}^S(u) = 1$ as ellipses to indicate heap elements; and we draw nodes having $\text{stack}^S(u) = 1$ as rectangles to indicate stack elements¹.

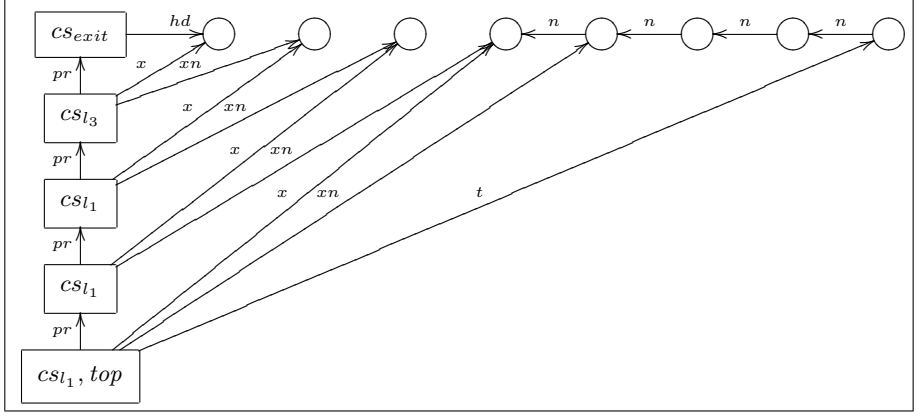


Fig. 2. The 2-valued structure S_2 which corresponds to the program state at l_2 in the **rev** procedure upon exit of the fourth recursive invocation of the **rev** procedure

Example 3.1 The 2-valued structure S_2 shown in Fig. 2 corresponds to the memory state at program point l_2 in the **rev** procedure upon exit from the fourth invocation of the **rev** procedure in the running example. The five rectangular nodes correspond to the activation records of the five procedure invocations. Note that our convention is that a stack grows downwards. The current activation record (of **rev**) is drawn at the bottom with *top* written inside. The three activation records (of **rev**) drawn above it correspond to pending invocations of **rev**.

The three isolated heap nodes on the left side of the figure correspond to the list elements pointed to by *x* in pending invocations of **rev**. The chain of five heap nodes to the right correspond to the (reversed) part of the original list. The last element in the list corresponds to the list element appended by **app** invoked just before l_2 in the current invocation of **rev**.

Notice that the *n* predicate is the only one that is specific to the linked list structure declared in Fig. 1(a). The remaining predicates would play a rule in the analysis of any data structure.

¹ This can be formalized alternatively using many sorted logics. We avoided that for the sake of simplicity, and for similarity with [15].

3.2 Consistent 2-Valued Structures

Some 2-valued structures cannot represent memory states, e.g., when a unary predicate g holds at two different nodes for a global variable g . A 2-valued structure is *consistent* if it can represent a memory state. It turns out that the analysis can be more precise by eliminating inconsistent 2-valued structures. Therefore, in Sect. 4.3 we describe a constructive method to check if a 2-valued structure is inconsistent and thus can be discarded by the analysis.

3.3 Kleene's 3-Valued Logic

Kleene's 3-valued logic is an extension of ordinary 2-valued logic with the special value of $\frac{1}{2}$ (unknown) for cases in which predicates could have either value, i.e., 1 (true) or 0 (false). Kleene's interpretation of the propositional operators is given in Fig. 3. We say that the values 0 and 1 are *definite values* and that $\frac{1}{2}$ is an *indefinite value*.

\wedge	0	1	$\frac{1}{2}$
0	0	0	0
1	0	1	$\frac{1}{2}$
$\frac{1}{2}$	0	$\frac{1}{2}$	$\frac{1}{2}$

\vee	0	1	$\frac{1}{2}$
0	0	1	$\frac{1}{2}$
1	1	1	1
$\frac{1}{2}$	$\frac{1}{2}$	1	$\frac{1}{2}$

\neg	
0	1
1	0
$\frac{1}{2}$	$\frac{1}{2}$

Fig. 3. Kleene's 3-valued interpretation of the propositional operators

3.4 Conservative Representation of Sets of Memory States via 3-Valued Structures

Like 2-valued structures, a *3-valued logical structure* S is also comprised of a universe U^S and an interpretation of the predicate symbols. However, for every predicate p of arity k , p^S is a function $p^S: (U^S)^k \rightarrow \{0, 1, \frac{1}{2}\}$, where $\frac{1}{2}$ explicitly captures unknown predicate values.

3-valued logical structures are also drawn as directed graphs. Definite values are drawn as in 2-valued structures. Binary indefinite predicate values are drawn as dotted directed edges. Also, we draw $p = \frac{1}{2}$ inside a node u when $p^S(u) = \frac{1}{2}$.

Let S^\natural be a 2-valued structure, S be a 3-valued structure, and $f: U^{S^\natural} \rightarrow U^S$ be a surjective function. We say that f *embeds* S^\natural *into* S if for every predicate p of arity k and $u_1, \dots, u_k \in U^{S^\natural}$, either $p^{S^\natural}(u_1, \dots, u_k) = p^S(f(u_1), \dots, f(u_k))$ or $p^S(f(u_1), \dots, f(u_k)) = \frac{1}{2}$. We say that S *conservatively represents all the 2-valued structures that can be embedded into it by some function f* . Thus, S can compactly represent many structures.

Nodes in a 3-valued structure that may represent more than one individual from a given 2-valued structure are called *summary nodes*. We use a designated unary predicate sm to maintain summary-node information. A summary node w has $sm^S(w) = \frac{1}{2}$, indicating that it may represent more than one node from 2-valued structures. These nodes are depicted graphically as dotted ellipses or rectangles. In contrast, if $sm^S(w) = 0$, then w is known to represent a unique node. We impose an additional restriction on embedding functions: only nodes with $sm^S(w) = \frac{1}{2}$ can have more than one node mapped to them by an embedding function.

Example 3.2 The 3-valued structure S_4 shown in Fig. 4 represents the 2-valued structure S_2 shown in Fig. 2. The dotted ellipse summary node represents all the eight list elements. The indefiniteness of the self n -edge results from the fact that there is an n -component pointer between each two successors and no n -component pointer between non-successors.

The dotted rectangle summary node represents the activation records from the second and third invocation of **rev**. The unary predicate cs_{l_1} drawn inside it indicates that it (only) represents activation records of **rev** that return to l_1 (i.e., recursive calls). The dotted x -edge from this summary node indicates that

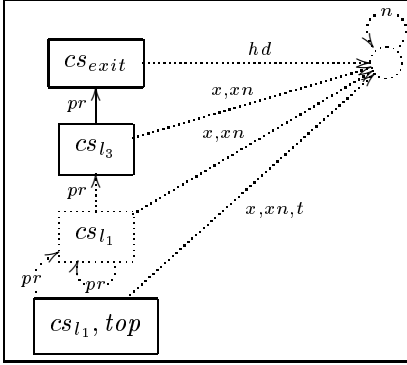


Fig. 4. The 3-valued structure S_4 which represents the 2-valued structure shown in Fig. 2

an invisible instance of x from the second or the third call may or may not point to one of the list elements. The rectangle at the top of Fig. 4 represents the activation record at the top of Fig. 2, which is the invocation of **main**. The second rectangle from the top in S_4 represents the second rectangle from the top in S_2 which is an invocation of **rev** from **main** (indicated by the occurrence of cs_{l_3} inside this node). The bottom rectangle in S_4 represents the bottom rectangle in S_2 , which is the current activation record (indicated by the occurrence of top inside this node). All other activation records are known not to be the

current activation record (i.e., the top predicate does not hold for these nodes) since top does not occur in either of them.

3.5 Expressing Properties via Formulae

Properties of structures can be extracted by evaluating formulae. We use first-order logic with transitive closure and equality, but without function symbols and constant symbols.² For example, the formula

$$\exists v_1, v_2 : \neg top(v_1) \wedge \neg top(v_2) \wedge v_1 \neq v_2 \wedge x(v_1, v) \wedge x(v_2, v) \quad (1)$$

expresses the fact that there are two different invisible instances of the parameter variable x pointing to the same list element v .

The Embedding Theorem (see [15, Theorem 3.7]) states that any formula that evaluates to a definite value in a 3-valued structure evaluates to the same value in all of the 2-valued structures embedded into that structure. The Embedding Theorem is the foundation for the use of 3-valued logic in static-analysis: It

² There is one non-standard aspect in our logic; $v_1 = v_2$ and $v_1 \neq v_2$ are indefinite in case v_1 and v_2 are the same summary node. The reason for this is seen shortly.

ensures that it is sensible to take a formula that—when interpreted in 2-valued logic—defines a property, and reinterpret it on a 3-valued structure S : The Embedding Theorem ensures that one must obtain a value that is conservative with regard to the value of the formula any 2-valued structure represented by S .

Example 3.3 Consider the 2-valued structure S_2 shown in Fig. 2. The formula (I) evaluates to 0 at all of the list nodes. In contrast, consider the 3-valued structure S_3 shown in Fig. 4. This formula (I) evaluates to $\frac{1}{2}$ at the dotted ellipse summary heap node. This is in line with the Embedding Theorem since $\frac{1}{2}$ is less precise than 0. However, it is not very precise since the fact that different invisible instances of x are never aliased is lost.

4 The Algorithm

In this section, we describe our shape-analysis algorithm for recursive programs manipulating linked lists. The algorithm iteratively annotates each program point with a set of 3-valued logical structures in a *conservative* manner, i.e., when it terminates, every 2-valued structure that can arise at a program point is represented by one of the 3-valued structures computed at this point. However, it may also conservatively include superfluous 3-valued structures.

Sect. 4.1 describes the properties of heap elements and local variables which are tracked by the algorithm. For ease of understanding, in Sect. 4.2, we give a high-level description of the iterative analysis algorithm. The actual algorithm is presented in Sect. 4.3.

4.1 Observing Selected Properties

To overcome the kind of imprecision described in Example 3.3, we introduce *instrumentation predicates*. These predicates are stored in each structure, just like the core predicates. The values of these predicates are derived from the core predicates, that is, every instrumentation predicate has a formula over the set of core predicates that defines its meaning. The instrumentation predicates that our interprocedural algorithm utilizes are described in Table 2, together with their informal meaning and their defining formula (other intraprocedural instrumentation predicates are defined in [15]).

The instrumentation predicates are divided into four classes, separated by double horizontal lines in Table 2: (i) Properties of heap elements with respect to visible variables, i.e., x and $r_{n,x}$. These are the ones originally used in [15]. (ii) Properties of heap elements with respect to invisible variables. These are \hat{x} and $r_{n,\hat{x}}$, which are variants of x and $r_{n,x}$ from the first class, but involve the invisible variables. The $sh_{\hat{x}}(v)$ predicate is motivated by Example 3.3. It is similar to the heap-sharing predicate used in [2,10,14,15]. (iii) Generic properties of an individual activation record. For example, $nn_x^S(u) = 1$ (nn for not NULL) in a 2-valued structure S indicates that the invisible instance of x that is stored in activation record u points to some list element. (iv) Properties across successive

Table 2. The instrumentation predicates used for the interprocedural analysis. Here x and y are generic names for local variables and parameters \mathbf{x} and \mathbf{y} of an analyzed function. The n^* notation used in the defining formula for $r_{n,x}(v)$ denotes the reflexive transitive closure of n

Predicate	Intended Meaning	Defining Formula
$x(v)$	The list element v is pointed to by the visible instance of \mathbf{x} .	$\exists v_1 : \text{top}(v_1) \wedge x(v_1, v)$
$r_{n,x}(v)$	The list element v is reachable by following \mathbf{n} -components from the visible instance of \mathbf{x} .	$\exists v_1, v_2 : \text{top}(v_1) \wedge x(v_1, v_2) \wedge n^*(v_2, v)$
$\hat{x}(v)$	The list element v is pointed to by an invisible instance of \mathbf{x} .	$\exists v_1 : \neg \text{top}(v_1) \wedge x(v_1, v)$
$r_{n,\hat{x}}(v)$	The list element v is reachable by following \mathbf{n} -component from an invisible instance of \mathbf{x} .	$\exists v_1, v_2 : \neg \text{top}(v_1) \wedge x(v_1, v_2) \wedge n^*(v_2, v)$
$sh_{\hat{x}}(v)$	The list element v is pointed to by more than one invisible instance of \mathbf{x} .	$\exists v_1, v_2 : v_1 \neq v_2 \wedge \neg \text{top}(v_1) \wedge \neg \text{top}(v_2) \wedge x(v_1, v) \wedge x(v_2, v)$
$nn_{\hat{x}}(v)$	The invisible instance of \mathbf{x} stored in the activation record v points to some list element.	$\exists v_1 : \neg \text{top}(v) \wedge x(v, v_1)$
$al_{x,y}(v)$	The invisible instances of \mathbf{x} and \mathbf{y} stored in the activation record v are aliased.	$\exists v_1 : \neg \text{top}(v) \wedge x(v, v_1) \wedge y(v, v_1)$
$al_{x,pr[y]}(v)$	The instance of \mathbf{x} stored in the activation record v is aliased with the instance of \mathbf{y} stored in v 's previous activation record.	$\exists v_1, v_2 : pr(v, v_1) \wedge x(v, v_2) \wedge y(v_1, v_2)$
$al_{x,pr[y] \rightarrow n}(v)$	The instance of \mathbf{x} stored in the activation record v is aliased with $\mathbf{y} \rightarrow \mathbf{n}$ for the instance of \mathbf{y} stored in v 's previous activation record.	$\exists v_1, v_2, v_3 : x(v, v_1) \wedge pr(v, v_2) \wedge y(v_2, v_3) \wedge n(v_3, v_1)$
$al_{x \rightarrow n, pr[y]}(v)$	$\mathbf{x} \rightarrow \mathbf{n}$ for the instance of \mathbf{x} stored in the activation record v is aliased with the instance of \mathbf{y} stored in v 's previous activation record.	$\exists v_1, v_2, v_3 : x(v, v_2) \wedge n(v_2, v_1) \wedge pr(v, v_3) \wedge y(v_3, v_1)$

recursive calls. For example, the predicate $al_{x,pr[y]}$ captures aliasing between \mathbf{x} at the callee and \mathbf{y} at the caller. The other properties are similar but also involve the \mathbf{n} component.

Example 4.1 The 3-valued structure S_5 shown in Fig. 5 also represents the 2-valued structure S_2 shown in Fig. 2. In contrast with S_4 shown in Fig. 4 in which all eight list elements are represented by one heap node, in S_5 they are represented by six heap nodes. The leftmost heap node in S_5 represents the leftmost list element in S_2 (which was originally the first list element). The fact that \hat{hd} is drawn inside this node indicates that it represents a list element pointed to by an invisible instance of \mathbf{hd} . This fact can also be extracted from S_5 by evaluating the \hat{hd} defining formula at this node, but this is not always the case, as we will now see: The second leftmost heap node is a summary node that represents both the second and third list elements from the left in S_2 . There is an indefinite x -edge into this summary node. Still, since \hat{x} is drawn inside it, every list element it represents must be pointed to by at least one invisible instance

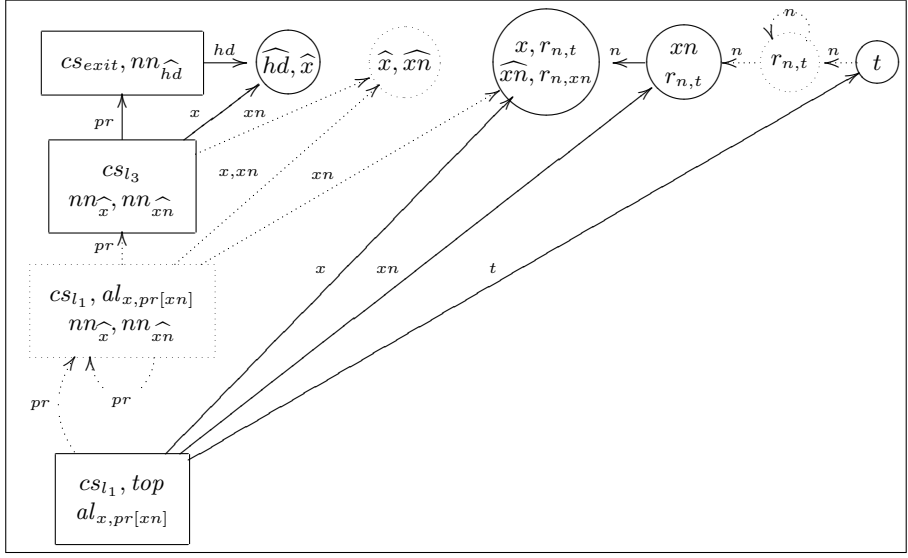


Fig. 5. The 3-valued structure S_5 represents the 2-valued structure shown in Fig. 2. For clarity, we do not show $r_{n,x}$ for nodes having the property x

of x . Therefore, the analysis can determine that this node does not represent storage locations that have been leaked by the program.

The other summary heap node (the second heap node from the right) represents the second and third (from the right) list elements of S_2 . Its incoming n edge is indefinite. Still, since $r_{n,t}$ occurs inside this node, we know that all the list elements it represents are reachable from t .

Note that the predicate $sh_{\hat{x}}$ does not hold for any heap node in S_5 . Therefore, no list element in any 2-valued structure that S_5 represents is pointed to by more than one invisible instance of the variable x . Note that the combination of $sh_{\hat{x}}(u) = 0$ (pointed to by at most one invisible instance of x) and $\hat{x}() = 1$ (pointed to by at least one invisible instance of x) allows determining that each node represented by a summary node u is pointed to by exactly one invisible instance of x (cf. the second leftmost heap node in S_5).

The stack elements are depicted in the same way as they are depicted by S_4 (see Example 3.2). Since $al_{x, pr[xn]}$ occurs inside the two stack nodes at the bottom, for every activation record v they represent, the instance of x stored in v is aliased with the instance of xn stored in the activation record preceding v .

Notice that the $r_{n,x}$, $r_{n,\hat{x}}$, $al_{x, pr[y] \rightarrow n}$, $al_{x \rightarrow n, pr[y]}$ predicates are the only instrumentation predicates specific to the linked list structure declared in Fig. 1(a). The remaining predicates would play a role in any analysis that would attempt to analyze the runtime stack.

4.2 The Best Abstract Transformer

This section provides a high level description of the algorithm in terms of the general abstract interpretation framework [3]. Conceptually, the most precise

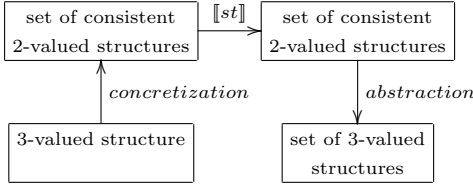


Fig. 6. The best abstract semantics of a statement st with respect to 3-valued structures. $\llbracket st \rrbracket$ is the operational semantics of st applied pointwise to every consistent 2-valued structure

(also called *best*) conservative effect of a program statement on a 3-valued logical structure S is defined in three stages shown in Fig. 6: (i) find each consistent 2-valued structure S^{\sharp} represented by S (*concretization*); (ii) apply the C operational semantics to every such structure S^{\sharp} resulting in a 2-valued structure $S^{\sharp'}$ and (iii) finally abstract each of the 2-valued structures $S^{\sharp'}$ by a 3-valued structure of bounded size (*abstraction*). Thus, the result of the statement is a set of 3-valued structures of bounded size.

The Abstraction Principle. The abstraction function is defined by a subset of the unary predicates, that are called *abstraction properties* in [15]. The abstraction of a 2-valued structure is defined by mapping all the nodes which have the same values for the abstraction properties into the same abstract node. Thus, the values of abstraction predicates remain the same in the abstracted 3-valued structures. The values of every other predicate p in the abstracted 3-valued structure are determined conservatively to yield an indefinite value whenever corresponding values of p in the represented concrete 2-valued structure disagree.

Example 4.2 The structure S_4 shown in Fig. 4 is an abstraction of S_2 shown in Fig. 2 when all of the unary core predicates are used as abstraction properties. For example, the activation records of the second and third recursive call to **rev** are both mapped into the summary stack node since they are both invisible activation records of invocations of **rev** from the same call-site (i.e., *top* does not hold for these activation records, but cs_{l_1} does). Also, all of the eight heap nodes are mapped to the same summary-node for which only the *heap* core predicate holds. The *pr*-edge into the stack node at the top of the figure is definite since there is only one node mapped to each of the edge’s endpoints. In contrast, the *hd*-edge emanating from the uppermost stack node must be indefinite in order for S_4 to conservatively represent S_2 . In S_2 the *hd* predicate holds for the uppermost stack node and the leftmost heap node, but it does not hold for any other heap node, and all heap nodes of S_2 are summarized into one summary heap node.

The structure S_5 shown in Fig. 5 is an abstraction of S_2 shown in Fig. 2 when the abstraction properties are all of the unary core and instrumentation predicates. Notice that nodes with different observed properties lead to different

instrumentation predicate values and thus are never represented by the same abstract node. Because the set of unary predicates is fixed, there can only be a constant number of nodes in an abstracted structure which guarantees that the analysis always terminates.

Analyzing Return Statements. How to retain precision when the analysis performs its abstract execution across a return statement is the key problem that we face. By exploiting the instrumentation predicates, our technique is capable of handling return statements quite precisely. This is demonstrated in the following example. For expository purposes we will explain the abstract execution of return statement in term of the best abstract transformer, described in Sect. 4.2. The actual method our analysis uses is discussed in Sect. 4.3.

Example 4.3 Let us exemplify the application of the return statement to the 3-valued structure S_5 shown in Fig. 5 following the stages of the *best* iterative algorithm described in Sect. 4.2.

Stage I–Concretization : Let S^\natural be one of the consistent 2-valued structures represented by S_5 . Let $k \geq 1$ be the number of activation records represented by the summary stack node in S_5 . Since S^\natural is a consistent 2-valued structure, the x parameter variable in each of these k activation records must point to one of the isolated list elements represented by the left summary heap node. This can be inferred by the following series of observations: the fact that the x variable in each of these activation records points to a list element is indicated by the presence of $nn_{\hat{x}}$ inside the stack summary node. The list elements pointed to by these variables must be represented by the left summary heap node since only one x -edge emanates from the summary stack node, and this edge enters the left summary heap node.

Let $m \geq 1$ be the number of list elements represented by the left summary heap node. Since \hat{x} occurs inside this node, each of the m list elements it represents must be pointed to by at least one invisible instance of x . Thus, $m \leq k$. However since $sh_{\hat{x}}$ does not occur inside this summary node, none of the m list elements it represents is pointed to by more than one invisible instance of x . Thus, we conclude that $m = k$.

Using the fact that $al_{x,pr[xn]}$ is drawn inside the two stack nodes at the bottom of Fig. 5, we conclude that the instance of x in each recursive invocation of **rev** is aliased with the instance of xn of **rev**'s previous invocation. Thus, each acceptable S^\natural looks essentially like the structure shown in Fig. 2, but with k isolated list elements not pointed to by **hd**, rather than two, and with some number of elements in the list pointed to by x .

Stage II–Applying the Operational Semantics: Applying the operational semantics of **return** to S^\natural (see Sect. 2) results in a (consistent) 2-valued structure $S^{\natural'}$. Note that the list element pointed to by the visible instance of x in $S^{\natural'}$ is not pointed to by any other instance of x , and it is not part of the reversed suffix. Thus $S^{\natural'}$ differs from S^\natural by having the top activation record of S^\natural removed from the stack and by having the activation record preceding it be the new *current* activation record.

Stage III–Abstraction.: Abstracting $S^{\sharp'}$ into a 3-valued structure may result, depending on k , in one of three possible structures. If $k > 2$ then the resulting structure is very similar to S_5 since the information regarding the number of remaining isolated list elements and invisible activation record is lost in the summarization. For $k = 1$ and $k = 2$ we have a consistent 2-valued structures with four and three activation records, respectively. Abstracting these structures results in no summary stack nodes, since the call-site of each non-current activation record is different. For $k = 1$ only one isolated list elements remains, thus it is not summarized. For $k = 2$ the two remaining isolated heap nodes are not merged since they are pointed to by different (invisible) local variables. For example, one of them is pointed to by `hd` and the other one is not.

Notice that if no instrumentation predicates correlating invisible variables and heap nodes are maintained, a conservative analysis cannot deduce that the list element pointed to by the visible instance of x in $S^{\sharp'}$ is not pointed to by another instance of this variable. Thus, the analysis must conservatively assume that future calls to `app` may create cycles. However, even when $al_{x,pr[xn]}$ is not maintained, the analysis can still produce fairly accurate results using only the $sh_{\hat{x}}$ and \hat{x} instrumentation predicates.

4.3 Our Iterative Algorithm

Unlike a hypothetical algorithm based on the best abstract transformer which explicitly applies the operational semantics to each of the (potentially infinite) structures represented by a three-valued structure S , our algorithm explicitly operates on S , yielding a set of 3-valued structures S' . By employing a set of judgements, similar in spirit to the ones described Example 4.3 our algorithm produces a set which conservatively represents all the structures that could arise after applying the `return` statement to each consistent 2-valued structure S represents. However, in general, the transformers used are conservative approximations of the best abstract transformer; the set of structures obtained may represent more 2-valued structures than those represented by applying the best abstract transformer. Our experience to date, reported in Sect. 5, indicates that it usually gives good results.

Technically, the algorithm computes the resulting 3-valued structure S' by evaluating formulae in 3-valued logic. When interpreted in 2-valued logic these formulae define the operational semantics. Thus, the Embedding Theorem guarantees that the results are conservative w.r.t a hypothetical algorithm based on the best abstract transformer. The update formulae for the core-predicates describing the operational semantics for call and return statements are given in Table 3.

Instead of calculating the instrumentation predicate values at the resulting structure by their defining formulae, which may be overly conservative, predicate-update formulae for instrumentation predicates are used. The formulae are omitted here for lack of space. The reader is referred to [15] for many examples of predicate-update formulae for instrumentation predicates and other operations used by the 3-valued logic framework to increase precision.

Table 3. The predicate-update formulae defining the operational semantics of the call and return statements for the core predicates. The value of each core predicate p after the statement executes, denoted by p' , is defined in terms of the core predicate values before the statement executes (denoted without primes). Core predicates that are not specified above are assumed to be unchanged, i.e., $p'(v_1, \dots) = p(v_1, \dots)$. There is a separate update formula for every local variable or parameter \mathbf{x} , and every label lb immediately preceding a procedure call. The predicate $new(v)$, used in the update formula of the $cs_{label}(v)$ predicates, holds only for the newly allocated activation record

label: call $f()$	return
$stack'(v) = stack(v) \vee new(v)$	$stack'(v) = stack(v) \wedge \neg top(v)$
$cs'_{label}(v) = cs_{label}(v) \vee new(v)$	$cs'_{lb}(v) = cs_{lb}(v) \wedge \neg top(v)$
$top'(v) = new(v)$	$top'(v) = \exists v_1 : top(v_1) \wedge pr(v_1, v)$
$pr'(v_1, v_2) = pr(v_1, v_2) \vee (new(v_1) \wedge top(v_2))$	$pr'(v_1, v_2) = pr(v_1, v_2) \wedge \neg top(v_1)$
	$x'(v_1, v_2) = x(v_1, v_2) \wedge \neg top(v_1)$

5 A Prototype Implementation

A prototype of the iterative algorithm sketched in Sect. 4.3 was implemented for a small subset of C, in particular we do not support mutual recursion. The main goal has been to determine if the results of the analysis are useful before trying to scale the algorithm to handle arbitrary C programs. In theory, the algorithm might be overly conservative and yield many indefinite values. This may lead to many “false alarms”. For example, the algorithm might have reported that every program point possibly leaked memory, performed a NULL-pointer dereference, etc. Fortunately, in Sect. 5.1 we show that this is not the case for the C procedures analyzed.

The algorithm was implemented using TVLA, a **Three-Valued-Logic Analyzer** which is implemented in Java [13]. TVLA is quite powerful but slow, and only supports intraprocedural analysis specified using low level logical formulae. Therefore, we implemented a frontend that generates TVLA input from a program in a subset of C. The instrumentation predicates described in Sect. 4.1 allow our frontend to treat call and return statements in the same way that intraprocedural statements are handled, without sacrificing precision in many recursive procedures. Our frontend also performs certain minimal optimizations not described here for lack of space.

5.1 Empirical Results

The analyzed C procedures together with the space used and the running time on a Pentium II 233 Mhz machine running Windows 2000 with JDK 1.2.2 are listed in Table 4. The analysis verified that indeed these procedures always return a linked list and contain no memory leaks and NULL-pointer dereferences. Verifying the absence of memory leaks is quite challenging for these procedures since it requires information about invisible variables as described in Sect. 4.1.

Table 4. The total number of 3-valued structures that arise during analysis and running times for the recursive procedures analyzed. The procedures are available at “<http://www.cs.technion.ac.il/~maon>”

Proc.	Description	# of Structs	Time (secs)
create	creates a list	219	5.91
delall	frees the entire list	139	13.10
insert	creates and inserts an element into a sorted list	344	38.33
delete	deletes an element from a sorted list	423	41.69
search	searches an element in a sorted list	303	8.44
app	adds one list to the end of another	326	42.81
rev	the running example (non recursive append)	829	105.78
rev_r	the running example (with recursive append)	2285	1028.80
rev_d	reverses a list with destructive updates	429	45.99

6 Conclusions, Limitations and Future Work

In this paper, we present a novel interprocedural shape analysis algorithm for programs that manipulate linked lists. The algorithm is more precise than existing shape analysis algorithms described in [2,10,14] for recursive programs that destructively update the program store. The precision of our algorithm can be attributed to the properties of invisible instances of local variables that it tracks. Particularly important seems to be the sharing properties of stack variables. Previous algorithms [2,10] either did not handle the case where multiple instances of the same local variable exist simultaneously, or only represented their potential values [14]. As we have demonstrated, in the absence enough information about the values of local variables, an analysis must make very conservative assumptions. These assumptions lead to imprecise results and performance deteriorates as well, since every potential value of a local variable must be considered.

We follow the approach suggested in [4,11] and summarize activation records in essentially the same way that linked list elements are summarized. By representing the call site in each activation record the analysis algorithm is capable of encoding the calling context, too. This approach bears some similarity to the *call-string* approach of [16], since it avoids propagating information to return sites that do not match the call site of the current activation record. In our case there is no need to put an arbitrary bound on the “length” of the call-string, the bounded representation is achieved indirectly by the summarization of activation records.

So far, our technique (and our implementation) analyzes small programs in a “friendly” subset of C. We plan to extend it to a larger subset of C, and to experiment with scaling it up to programs of realistic size. One possible way involves first running a cheap and imprecise pointer-analysis algorithm, such as the flow-insensitive points-to analysis described in [17], before proceeding to our quite precise but expensive analysis. We focused this research on linked lists, but, plan to also investigate tree-manipulation programs.

Finally, our analysis is limited by its fixed set of predefined “library” properties. This makes our tool easy to use since it is fully automatic and does not require any user intervention, e.g., a specification of the program. However, this is a limitation because the analyzer produce poor results for program in which other properties are the important distinctions to track.

Acknowledgments. We are grateful for the helpful comments and the contributions of N. Dor, O. Grumberg, T. Lev-Ami, R. Wilhelm, T. Reps and E. Yahav.

References

1. U. Assmann and M. Weinhardt. Interprocedural heap analysis for parallelizing imperative programs. In W. K. Giloi, S. Jähnichen, and B. D. Shriver, editors, *Programming Models For Massively Parallel Computers*, September 1993.
2. D.R. Chase, M. Wegman, and F. Zadeck. Analysis of pointers and structures. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 296–310, 1990.
3. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Symp. on Princ. of Prog. Lang.*, pages 269–282, New York, NY, 1979. ACM Press.
4. A. Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *Symp. on Princ. of Prog. Lang.*, 1990.
5. N. Dor, M. Rodeh, and M. Sagiv. Checking cleanness in linked lists. In *SAS’00, Static Analysis Symposium*. Springer, 2000.
6. R. Ghiya and L. Hendren. Putting pointer analysis to work. In *Symp. on Princ. of Prog. Lang.*, New York, NY, 1998. ACM Press.
7. R. Ghiya and L.J. Hendren. Is it a tree, a dag, or a cyclic graph? In *Symp. on Princ. of Prog. Lang.*, New York, NY, January 1996. ACM Press.
8. L. Hendren. *Parallelizing Programs with Recursive Data Structures*. PhD thesis, Cornell Univ., Ithaca, NY, Jan 1990.
9. L. Hendren, J. Hummel, and A. Nicolau. Abstractions for recursive pointer data structures: Improving the analysis and the transformation of imperative programs. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 249–260, June 1992.
10. N.D. Jones and S.S. Muchnick. Flow analysis and optimization of Lisp-like structures. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 4. Prentice-Hall, Englewood Cliffs, NJ, 1981.
11. N.D. Jones and S.S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Symp. on Princ. of Prog. Lang.*, pages 66–74, New York, NY, 1982. ACM Press.
12. J.R. Larus and P.N. Hilfinger. Detecting conflicts between structure accesses. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 21–34, 1988.
13. T. Lev-Ami and M. Sagiv. TVLA: A framework for Kleene based static analysis. In *SAS’00, Static Analysis Symposium*. Springer, 2000.
14. M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *Trans. on Prog. Lang. and Syst.*, 20(1):1–50, Jan 1998.
15. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Symp. on Princ. of Prog. Lang.*, 1999.
16. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–234. Prentice-Hall, Englewood Cliffs, NJ, 1981.
17. B. Steensgaard. Points-to analysis in almost-linear time. In *Symp. on Princ. of Prog. Lang.*, pages 32–41, 1996.

Design-Driven Compilation

Radu Rugina and Martin Rinard

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
{rugina, rinard}@lcs.mit.edu

Abstract. This paper introduces *design-driven compilation*, an approach in which the compiler uses design information to drive its analysis and verify that the program conforms to its design. Although this approach requires the programmer to specify additional design information, it offers a range of benefits, including guaranteed fidelity to the designer's expectations of the code, early and automatic detection of design non-conformance bugs, and support for local analysis, separate compilation, and libraries. It can also simplify the compiler and improve its efficiency. The key to the success of our approach is to combine high-level design specifications with powerful static analysis algorithms that handle the low-level details of verifying the design information.

1 Introduction

Compilers have traditionally operated on the source code alone, utilizing no other sources of information about the computation or the way the designer intends it to behave. But the source code is far from a perfect source of information for the compiler. We focus here on two drawbacks. First, the code is designed for efficient execution, not presentation of useful information. The information is therefore often *obscured* in the program: even though the code implicitly contains the information that the compiler needs to effectively compile the program, the information may be difficult or (for all practical purposes) impossible to extract. Second, the source code may be *missing*, either because it is shipped in unanalyzable form or because it has yet to be implemented.

The thesis of this paper is that augmenting the source code with additional design information can ameliorate or even eliminate these problems. The result is a significant increase in the *reach* of the compiler (its ability to extract information about the program and use this information to transform the program) and the *fidelity* with which the compilation matches the designer's intent. We call this new compilation paradigm *design-driven compilation*, because the design information drives the compiler's search for information.

We believe the key to the success of this approach is an effective division of labor between the designer and the compiler. The design information should take the form of intuitive, high-level properties that any designer would need to know to design the computation. This information must be augmented by

powerful analysis algorithms that automate the low-level details of verifying the design information and applying the design.

1.1 Design Conformance

The design information specifies properties that the program is intended to fulfill. This information is implicitly encoded in the program, and the compiler would otherwise need sophisticated program analysis techniques to extract it. But the explicit (and redundant) specification of design information provides a range of advantages. First, it clearly states the programmer's intent, and non-conformance to the design can be caught early by the compiler. Second, using design information at the procedure level provides modularity and enables local analysis, separate compilation, and library support. Third, it is easier for the compiler to check the design information than to extract it, making the compiler structure simpler and making the compiler more efficient.

Design information for procedures can be expressed using *procedure interfaces*. The interface for a procedure specifies the effects of that procedure with respect to a given abstraction. In this paper we present two kinds of procedure interfaces. Pointer interfaces specify how procedures change the points-to information. Region interfaces specify the regions of memory that the entire computation of each procedure accesses.

It is not practical to expect the programmer provide detailed information within procedures. But the correctness of the design information depends crucially on the low-level actions of the code within each procedure. Our approach therefore *combines design specifications with program analysis*. Design information specifies high-level properties of the program. Static analysis automatically extracts the low-level properties required to verify that the program conforms to its design. Design conformance is an attractive alternative to static analysis alone because of its range of benefits:

1. Fidelity:

- *Faithful Compilation*: The design information enables the compiler to generate parallel programs that faithfully reflect the designer's high-level expectations.
- *Enhanced Code Reliability*: Verifying that the code conforms to its design eliminates many potential programming errors. Subtle off-by-one errors in array index and pointer arithmetic calculations, for example, often cause the program to violate its region interface design.
- *Enhanced Design Utility*: Our approach guarantees that the program conforms to its design. Designers, programmers, and maintainers can therefore rely on the design to correctly reflect the behavior of the program, enhancing the utility of the design as a source of information during the development and maintenance phases.

2. Modularity:

- *Local Analysis*: The design information enables the compiler to use a *local* instead of a *global* analysis — each procedure is analyzed independently of all other procedures.

- *Separate Compilation*: The design information enables the compiler to fully support the separate analysis and compilation of procedures in different files or modules, and to fully support the use of libraries that do not export procedures in analyzable form.
- *Improved Development Methodology*: The design information allows the compiler to analyze incomplete programs as they are under development. The programmer can therefore start with the design information, then incrementally implement each procedure. At each step during the development, the analysis uses the design information to check that the current code base correctly conforms to its design. The overall result is early detection of any flaws in the design and an orderly development of a system that conforms to its design.
- *Enhanced Interface Information*: In effect, our approach extends the type system of the language to include additional information in the type signature of each procedure. The additional information formalizes an additional aspect of the procedure’s interface, making it easier for engineers to understand and use the procedures.

3. Simplicity:

The availability of design information as procedure interfaces significantly simplifies the structure of the compiler. It eliminates the interprocedural analysis, and replaces it with the much simpler task of verifying the procedure interfaces. Improvements include increased confidence in the correctness of the compiler, a reduction in the implementation time and complexity, and increased compiler efficiency.

1.2 Automatic Parallelization of Divide and Conquer Programs

We have applied design-driven compilation to a challenging problem: the automatic parallelization of divide and conquer programs [4]. The inherent parallelism and good cache locality of divide and conquer algorithms make them a good match for modern parallel machines, with excellent performance on a range of problems [210,63].

The tasks in divide and conquer programs often access disjoint regions of the same array. To parallelize such a program, the compiler must precisely characterize the regions of memory that the complete computation of each procedure accesses. But it can be quite difficult to extract this information automatically. Divide and conquer programs use recursion as their primary control structure, invalidating standard approaches that analyze loops. They also tend to use dynamic memory allocation to match the sizes of the data structures to the problem size. The data structures are then accessed via pointers and pointer arithmetic, which complicates the analysis of the memory regions accessed by the computation of each procedure.

In our approach, the designer provides two kinds of design information: pointer information that summarizes how each procedure affects the points-to relationships, and region information that summarizes regions of memory that each procedure accesses. The compiler uses intraprocedural analysis algorithms

```

1 void sort(int *d, int *t, int n) {
2     int *d1, *d2, *d3, *d4, *d5;
3     int *t1, *t2, *t3, *t4;
4     if (n < CUTOFF) {
5         insertionsort(d, d+n);
6     } else {
7         d1 = d; t1 = t;
8         d2 = d1 + n/4; t2 = t1 + n/4;
9         d3 = d2 + n/4; t3 = t2 + n/4;
10        d4 = d3 + n/4; t4 = t3 + n/4;
11        d5 = d4+(n-3*(n/4));
12
13        sort(d1, t1, n/4);
14        sort(d2, t2, n/4);
15        sort(d3, t3, n/4);
16        sort(d4, t4, n-3*(n/4));
17
18        merge(d1, d2, d2, d3, t1);
19        merge(d3, d4, d4, d5, t3);
20
21        merge(t1, t3, t3, t1+n, d);
22    }
23 }

24 void merge(int *l1, int *h1,
25            int *l2, int *h2, int *d) {
26     while ((l1 < h1) && (l2 < h2))
27         if (*l1 < *l2) *d++ = *l1++;
28         else *d++ = *l2++;
29     while (l1 < h1) *d++ = *l1++;
30     while (l2 < h2) *d++ = *l2++;
31 }

32 void insertionsort(int *l, int *h) {
33     int *p, *q, k;
34     for (p = l+1; p < h; p++) {
35         k = *p;
36         for (q = p-1; l <= q && k < *q; q--)
37             *(q+1) = *q;
38         *(q+1) = k;
39     }
40 }

41 void main() {
42     int n, *data, *temp;
43     scanf("%d", &n);
44     if (n > 0) {
45         data = (int*) malloc(n*sizeof(int));
46         temp = (int*) malloc(n*sizeof(int));
47         /* code to initialize the array */
48         sort(data, temp, n);
49         /* code that uses the sorted array */
50     }
51 }

```

Fig. 1. Divide and Conquer Sorting Example

to verify the design information, then uses the verified information to parallelize the program.

2 Example

Figure 1 presents a recursive, divide and conquer merge sort program. The `sort` procedure on line 1 takes an unsorted input array `d` of size `n`, and sorts it, using the array `t` (also of size `n`) as temporary storage. In the divide part of the algorithm, the `sort` procedure divides the two arrays into four sections and, in lines 13 through 16, calls itself recursively to sort the sections. Once the sections have been sorted, the combine phase in lines 18 through 21 produces the final sorted array. It merges the first two sorted sections of the `d` array into the first half of the `t` array, then merges the last two sorted sections of `d` into the last half of `t`. It then merges the two halves of `t` back into `d`. The base case of the algorithm uses the insertion sort procedure in lines 32 through 40 to sort small sections.

```

merge(int *l1, int *h1,
      int *l2, int *h2,
      int *d) {
  context {
    input , output :
    l1 -> main:alloc1,
    h1 -> main:alloc1,
    l2 -> main:alloc1,
    h2 -> main:alloc1,
    d -> main:alloc2
  }
  context {
    input , output :
    l1 -> main:alloc2,
    h1 -> main:alloc2,
    l2 -> main:alloc2,
    h2 -> main:alloc2,
    d -> main:alloc1
  }
}

insertionsort(int *l, int *h) {
  context {
    input , output :
    l -> main:alloc1,
    h -> main:alloc1
  }
}

sort(int *d, int *t, int n) {
  context {
    input , output :
    d -> main:alloc1,
    t -> main:alloc2
  }
}

```

Fig. 2. Points-To Design Information

```

merge(int *l1, int *h1,
      int *l2, int *h2,
      int *d) {
  reads [l1,h1-1], [l2,h2-1];
  writes [d,d+(h1-l1)+(h2-l2)-1];
}

insertionsort(int *l, int *h) {
  reads and writes [l,h-1];
}

sort(int *d, int *t, int n) {
  reads and writes [d,d+n-1],
                  [t,t+n-1];
}

```

Fig. 3. Access Region Design Information

2.1 Design Information

There are two key pieces of design information in this computation: information about where each pointer variable points to during the computation, and information about the regions of the arrays that each procedure accesses. Our design language enables designers to express both of these pieces of information, enhancing the transparency of the code and enabling the parallelization transformation described below in Section 2.4.

Figure 2 shows how the designer specifies the pointer information in this example. For each procedure, the designer provides a set of *contexts*. Each context is a pair of *input* points-to edges and *output* points-to edges. The input set of edges represents the pointer aliasing information at the beginning of the procedure, and the output set of edges represents that information at the end of the procedure for that given input. Therefore, each context represents a partial transfer function: it describes the effect of the execution of the procedure for a given input points-to information. The pointer analysis takes place at the

granularity of *allocation blocks*. There is one allocation block for each static or dynamic allocation site. The design information identifies each allocation site using the name of the enclosing procedure and a number that specifies the allocation site within the procedure. In our example, the input and the output information are the same for all contexts, which means that all procedures in our example have identity transfer functions.

Figure 3 shows how the designer specifies the accessed memory regions in the example. The regions are expressed using *memory region expressions* of the form $[l, h]$, which denotes the region of memory between l and h , inclusive. These regions are expressed symbolically in terms of the parameters of each procedure. This symbolic approach is required because during the course of a single computation, the procedure is called many times with many different parameter values.

As the example reflects, both pointer and access region specifications build on the designer’s conception of the computation. The specification granularity matches the granularity of the logical decomposition of the program into procedures, with the specifications formalizing the designer’s intuitive understanding of the regions of memory that each procedure accesses.

2.2 Pointer Design Conformance

The compiler verifies that the program conforms to its pointer design as follows. For each procedure and each context, the compiler performs an intraprocedural, flow-sensitive pointer analysis of the body of the procedure. At call sites, the compiler matches the current points-to information with the input information of one of the contexts of the callee procedure. It uses the output information from matched context to compute the points-to information after the call. If no matching context is found, the pointer design conformance fails. In our example, during the intraprocedural analysis of procedure **sort**, the compiler matches the current points-to information at lines 18 and 19 with the first context for **merge**, and the points-to information at line 21 with the second context of **merge**. The **sort** and **insertionsort** procedures each have only one context, and the compiler successfully matches the context at each call to one of these two procedures. Note that the pointer design information directly gives the fixed-point solution for the recursive procedure **sort**. The compiler only checks that this solution is a valid solution.

2.3 Access Region Design Verification

The compiler verifies the access region design in two steps. The first step is an intraprocedural analysis, called *bounds analysis*, that computes lower and upper bounds for each pointer and array index variable at each program point. This bounds information for variables immediately translates into the regions of memory that the procedure directly accesses via load and store instructions.

The second step is the verification step. To verify that for each procedure the design access regions correctly reflect the regions of memory that the whole

computation of the procedure accesses, the compiler checks the following two conditions:

1. the design access regions for each procedure include the regions directly accessed by the procedure, and
2. the design access regions for each procedure include the regions accessed by its invoked procedures.

In our example, for procedure `insertionsort` the compiler uses static analysis to compute the bounds of local pointer variables `p` and `q` at each program point, and then uses these bounds at each load and store in the procedure to detect that `insertionsort` directly reads and writes the memory region $[1, h-1]$. The compiler easily checks that this region is included in the access region for `insertionsort` from the design specification. The verification proceeds similarly for procedure `merge`.

For the recursive procedure `sort`, the design specifies two access regions: $[d, d+n-1]$ and $[t, t+n-1]$. For the first recursive call to `sort` at line 13, the compiler uses the design access regions to derive the access regions for this particular call statement: $[d, d+n/4-1]$ and $[t, t+n/4-1]$. It then verifies that these access regions are included in the design access regions for `sort`: $[d, d+n/4-1] \subseteq [d, d+n-1]$ and $[t, t+n/4-1] \subseteq [t, t+n-1]$. The compiler uses a similar reasoning to verify that all the call statements in `sort` comply with the design specification, and concludes that the implementation of `sort` conforms to its design.

A negative result in the verification step often indicates a subtle array addressing bug. For example, changing the `<` operator to `<=` in lines 26, 29, or 30 causes the compiler to report that the procedure does not conform to its design, as does changing `1+1` to `1` on line 34.

2.4 Parallelization

There are two sources of concurrency in the example: the four recursive calls to the `sort` procedure can execute in parallel, and the first two calls to the `merge` procedure can execute in parallel. Executing these calls in parallel leads to a recursively generated form of concurrency in which each parallel sort task, in turn, recursively generates additional parallel tasks.

The compiler recognizes this form of concurrency by comparing pairs of region expressions from different procedure calls and statements to determine if they are independent. Two region expressions are independent if they denote disjoint (non-overlapping) regions of memory or they both denote regions that are read. In our example, the four recursive calls to the `sort` procedure access independent region expressions and can execute in parallel with each other, as do the first two calls to the `merge` procedure.

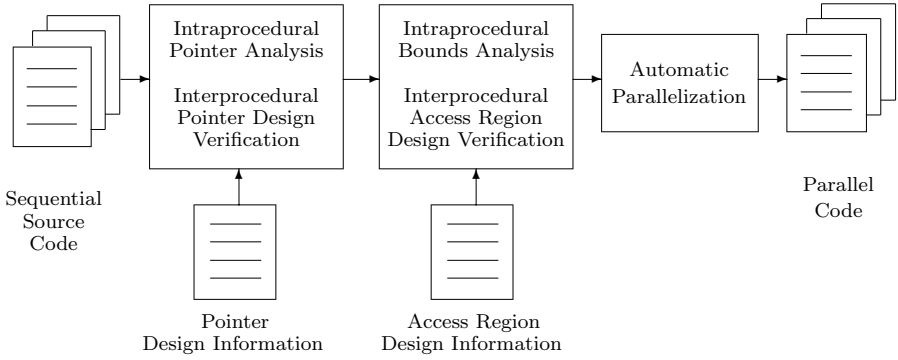


Fig. 4. The Structure of the Compiler

3 Structure of the Compiler

Figure 4 presents the general structure of the compiler. The compiler first uses a context-sensitive, flow-sensitive, and intraprocedural pointer analysis to verify the pointer design information [15]. It next uses an intraprocedural algorithm to verify the design access regions. Finally, it uses the verified design information to automatically parallelize the computation. We next discuss the static analysis and design verification algorithms in more detail.

3.1 Pointer Analysis and Design Verification

The intraprocedural pointer analysis extracts points-to information at each program point¹. It represents points-to information using points-to graphs [5]. The nodes in this graphs are program variables, and the edges in the graph represent points-to relations between variables. The compiler also handles dynamically allocated objects, distinguishing between them based on their allocation site. The compiler uses special variables, called ghost variables, to represent variables on the stack that are not in the scope of the currently analyzed procedure, but are accessible via parameters or global pointers from the current procedure.

In the intraprocedural analysis, our compiler uses a flow-sensitive, pointer analysis algorithm [15]. It uses a standard dataflow analysis approach, with specific analysis rules for pointer assignments via copy, load, and store statements. It uses the design information to compute transfer functions for procedure calls. Each context in the design specifies a partial transfer function for that procedure [18], and the analysis directly uses the output of the context whose input matches the current points-to information at the call.

In the specifications, points-to graphs are represented as sets of edges between program variables. Dynamically allocated objects are specified using both

¹ Although in this section we present how we verify points-to information, similar techniques can be employed for any other dataflow information.

the name of the enclosing procedure that allocates them and a number indicating which allocation site within that procedure it is referring to. For instance `main:alloc1` represents an object allocated at the first dynamic allocation site in procedure `main`. Ghost variables are specified using their type, for instance `ghost(int[10])` describes an array of integers allocated on stack, accessible, but not visible to the current procedure.

When the analysis of each context completes, the compiler checks that the analysis result for that context matches the corresponding output points-to information from the design. If the design is verified, the pointer analysis results at each program point can be safely used in the following stages of the compiler.

3.2 Access Region Analysis and Design Verification

The compiler next uses the same approach of combining static analysis and design verification to derive symbolic regions of memory that the whole computation of each procedure accesses. The lower and upper bounds of these regions are symbolic polynomial expressions in the initial values of the parameters of the enclosing procedure. Both the analysis and the design verification presented in this section separately keep track of read and written regions of memory.

The symbolic access region in the specification represent regions within allocation blocks. The general format of an access region within an allocation block relative to a procedure f is: $[p : l, h]$ where l and h are symbolic expressions in the initial parameters of f and p is a pointer variable. This denotes a region with lower bound l and upper bound h within all the allocation blocks pointed to by p , at the beginning of f , for all points-to contexts of f . If the lower bound l is a simple symbolic expression consisting of a single term equal to p , then we use the shortcut notation $[l, h]$. All the specifications in our example from Figure 11 use this shortcut notation.

The compiler extracts and verifies the access region information as follows:

1. **Static Analysis:** The compiler first performs an intraprocedural analysis, called *bounds analysis*, to extract lower and upper bounds for each pointer and array index variable at each program point. The algorithm is presented in detail in [16]. The compiler then uses the extracted bounds information for pointers and array indices to derive access regions for each memory access (i.e. each load and store) in the program. The compiler finally combines these access regions for loads and stores and derives the regions of memory that each procedure *directly* accesses.
2. **Design Verification:** The compiler next uses the intraprocedural access region results to verify that the access region design specification correctly characterizes the memory regions that the whole execution of each procedure accesses. To verify the safety of the design access regions, the compiler checks two conditions. First, the design access regions of each procedure should include the access regions directly accessed by that procedure. Second, the design access regions of each procedure should include the design access regions of all its invoked procedures. If both conditions hold for all procedures, the design is verified. Otherwise, access design verification fails.

During the verification process at call statements, the compiler uses the access regions of the callee to derive an access region for the call statement. But the analysis of the callee produces a result in terms of the initial values of the callee’s parameters. The result for the caller must be expressed in terms of the caller parameters, not the callee parameters. The *symbolic unmapping* algorithm performs this change of analysis domain. A detailed description and a formal definition of symbolic unmapping is given in [16]. The idea behind symbolic unmapping of an access region is to replace the callee’s parameters in the region bounds with the actual parameters at the call site, and then use the bounds information at the call site to express the access region in terms of the initial values of the caller’s parameters.

Once it verifies the access region design information, the compiler can safely use it to detect sequences of independent calls and generate parallel code to execute them concurrently.

4 Experimental Results

We have implemented a compiler that combines the static analysis algorithms and the design verification algorithms presented in this paper. This compiler was implemented using the SUIF compiler infrastructure [1]. We implemented all of the analyses, including the pointer analysis, from scratch starting with the standard SUIF distribution. Our compiler generates parallel code in Cilk [7], a parallel dialect of C.

We present experimental results for two recursive sorting programs (Quicksort and Mergesort), a divide and conquer blocked matrix multiply (BlockMul), a divide and conquer LU decomposition (LU), and a scientific computation (Heat). We would like to emphasize the challenging nature of the programs in this benchmark set. Most of them contain multiple mutually recursive procedures, and have been heavily optimized by hand to extract the maximum performance. As a result, they heavily use low-level C features such as pointer arithmetic and casts.

4.1 Design Conformance

Using the approach presented on this paper, the compiler successfully verified that all the benchmarks comply to both their pointer design and to their access region design. The compiler used the extracted intraprocedural pointer information and access region information to carry out the design verification process.

4.2 Design Information Size and Complexity

We compare the complexity of the access region design as opposed to the program by computing the ratio of the number of bytes in the program divided by the number of bytes in the design. Table 1 separately presents the results for pointer design specifications and access region design specifications, which show that our set of benchmark programs is between 6 and 27 times larger than their designs.

Table 1. Ratio of Program Size to Design Size

Program	Program to Pointer Design Ratio	Program to Region Access Design Ratio
Quicksort	10	15
Mergesort	6	14
Heat	10	12
BlockMul	15	27
LU	11	12

Table 2. Pointer Analysis Running Times (in seconds)

	Pointer Analysis Alone	Pointer Analysis and Design Information
Quicksort	0.02	0.01
Mergesort	0.05	0.04
Heat	0.13	0.09
BlockMul	3.45	1.84
LU	0.30	0.15

Our own qualitative assessment of the design information is that it is very easy for the designer to provide, in part because it is a natural, intuitive extension of the procedure interface, and in part because it is so small in comparison with the programs.

4.3 Compiler Complexity and Efficiency

Table 2 shows the running times of the pointer analysis phase using combined program analysis and design information compared to program analysis alone. The availability of design information can produce speedups up to a factor of 2 for our set of benchmarks. For the access region phase the running times were roughly the same with and without design information. Here the bottleneck was the intraprocedural analysis, which is executed in both cases.

The availability of design information significantly decreased both the complexity and the implementation time of the analysis. Compared to the implementation in our previous work for the automatic parallelization of divide and conquer algorithms [14, 16], the design-based approach presented in this paper eliminated sophisticated interprocedural algorithms based on fixed-point algorithms or on reductions to linear programs. These complex analyses were replaced by the simple design verification algorithms presented in the current paper. This reduction in compiler complexity also translated in a reduction of implementation time from the order of months to the order of days for the replaced sections of the compiler.

Table 3. Absolute Speedups

Programs	Number of Processors				
	1	2	4	6	8
Quicksort	1.00	1.99	3.89	5.68	7.36
Mergesort	1.00	2.00	3.90	5.70	7.41
Heat	1.03	2.02	3.89	5.53	6.83
BlockMul	0.97	1.86	3.84	5.70	7.54
LU	0.98	1.95	3.89	5.66	7.39

4.4 Automatic Parallelization

Our analysis was able to automatically parallelize all of the applications. We ran the benchmarks on an eight processor Sun Ultra Enterprise Server. Table 3 presents the speedups. These speedups are given with respect to the sequential versions, which execute with no Cilk overhead. For Heat, the Cilk program running on one processor runs faster than the sequential version, in which case the absolute speedup is above one for one processor. We ran Quicksort and Mergesort on a randomly generated file of 8000000 numbers and BlockMul and LU on a 1024 by 1024 matrix.

5 Related Work

5.1 Access Specifications

The concept of allowing programmers to specify how constructs access data is a continually arising subtheme in programming languages. The effect system in FX/87, for example, allows programmers to specify the *effects* of each procedure, i.e., the regions that it accesses [8]. The type checking algorithm is extended to statically verify that the specified effects correctly reflect the accesses of the procedure. Access declarations in Jade allow programmers to specify how tasks access shared objects [13]. The access declarations are used to parallelize the program, and are dynamically checked by the Jade run-time system. In both Jade and FX/87, the specifications operate at the granularity of complete objects — there is no way to specify that a procedure or task accesses part of an array or object. The access specifications in this paper, on the other hand, operate at the granularity of subregions of the accessed arrays. They therefore enable the compiler to recognize (and parallelize) procedure calls that access disjoint regions of the same array.

5.2 Interprocedural Array Region Analysis

Several researchers have developed systems that automatically characterize the array regions that procedures access. The first systems were designed to analyze

scientific programs with loop nests that manipulate dense matrices using affine access functions [17,12,11]. These systems use the loop bounds and the array index expressions to derive the array regions that each procedure accesses. They then propagate accessed array regions from callees to callers to derive the regions accessed by the complete execution of each procedure. Researchers have recently generalized this approach for recursive procedures that access data via pointers [14,9]. An issue is maintaining precision in the face of the fixed-point computations used to analyze recursive procedures. Our recent generalization of the intraprocedural approach presented in Section 3 to accurately analyze recursive procedures without fixed-points eliminates this particular problem [16].

The bottom line is that it is possible, in principle, to attack the problem of parallelizing divide and conquer programs without design information in the form of access regions. We nevertheless see such design information and design conformance as playing a desirable role in this context, for the following reasons:

- **Simplicity:** Access regions enable the compiler to apply a simple intraprocedural algorithm. Eliminating the interprocedural analysis significantly simplifies the structure of the compiler and its analysis algorithms. Improvements include increased confidence in the correctness of the compiler and a reduction in the implementation time and complexity.
- **Independence:** Access regions enable the compiler to analyze and compile each procedure independently of all other procedures. The analysis is therefore more efficient and scalable since it does not have to perform a global analysis. The design information also enables the compiler to support separate compilation, unanalyzable libraries, and missing code in programs under development.
- **Development Improvements:** Access regions are an intuitive formalization of a key aspect of the design of the program. They are easy for designers to provide, in part because they simply crystalize information that the designer must already have available to successfully design the algorithm, and in part because the designer provides only a small amount of information at procedure boundaries. They also provide a natural extension to standard procedure interfaces, improving the transparency of the code and giving clients additional information about the interface of the procedure. Finally, they can help the debugging process: subtle array addressing bugs often show up as violations of the declared access regions.

6 Future Work

Information about the ranges of pointer and array index variables can be used for purposes other than automatic parallelization. For example, many security problems are caused by incorrect programs that an attacker can coerce into violating its array bounds. We believe that enabling the designer to explicitly state the array referencing expectations inherent in the design would help developers produce software without these problems. Developers could therefore use our

approach to verify that the program has no security vulnerabilities caused by array bounds violations.

Languages such as Java use dynamic checks to eliminate array bounds violations. The advantage is that array bounds violations are caught before they corrupt the system; the disadvantage is the overhead of performing the array bounds checks dynamically. And an array bounds violation is still an error, and typically causes the program to fail. By statically verifying that programs do not violate their array bounds, our proposed techniques can both eliminate dynamic array bounds check overhead and improve the reliability of the delivered software.

7 Conclusion

This paper presents design-driven compilation, a technique for using design information to improve the analysis and compilation of the program. Design-driven compilation uses design information to drive its analysis and verify that the program conforms to its design. The main advantages of design-driven compilation are the fidelity to the designer's expectations, analysis modularity, and simplicity and efficiency of the compiler. We have applied this approach to the problem of automatic parallelization of divide and conquer programs. Our results show that the design information is small compared to the program, works well with the designer's intuitive conception of the structure, decreases the complexity of the compiler while increasing its efficiency, and enables the compiler to generate parallel code with excellent performance.

In the future, we anticipate that design conformance will become an increasingly important. In addition to enabling the compiler to better analyze both complete and incomplete programs, it will also help designers and implementors deliver more reliable programs that are guaranteed to conform to their designs. We anticipate that this automatically checked connection between the design and the implementation will significantly increase the role that formal designs play during the implementation and maintenance phases, reducing the cost of these phases and increasing the robustness of the delivered software.

Acknowledgements. We would like to thank Daniel Jackson for many interesting conversations regarding design conformance.

References

1. S. Amarasinghe, J. Anderson, M. Lam, and C. Tseng. The SUIF compiler for scalable parallel machines. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, February 1995.
2. R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995. ACM, New York.

3. S. Chatterjee, A. Lebeck, P. Patnala, and M. Thottethodi. Recursive array layouts and fast matrix multiplication. In *Proceedings of the 11th Annual ACM Symposium on Parallel Algorithms and Architectures*, Saint Malo, France, June 1999.
4. T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introductions to Algorithms*. The MIT Press, Cambridge, Mass., Cambridge, MA, 1990.
5. Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the SIGPLAN '94 Conference on Program Language Design and Implementation*, Orlando, FL, June 1994.
6. J. Frens and D. Wise. Auto-blocking matrix-multiplication or tracking BLAS3 performance from source code. In *Proceedings of the 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Las Vegas, NV, June 1997.
7. M. Frigo, C. Leiserson, and K. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the SIGPLAN '98 Conference on Program Language Design and Implementation*, Montreal, Canada, June 1998.
8. D. Gifford, P. Jouvelot, J. Lucassen, and M. Sheldon. FX-87 reference manual. Technical Report MIT/LCS/TR-407, Laboratory for Computer Science, Massachusetts Institute of Technology, September 1987.
9. M. Gupta, S. Mukhopadhyay, and N. Sinha. Automatic parallelization of recursive procedures. Technical report, IBM T. J. Watson Research Center, 1999.
10. F. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM Journal of Research and Development*, 41(6):737–755, November 1997.
11. M.W. Hall, S.P. Amarasinghe, B.R. Murphy, S. Liao, and M.S. Lam. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995. IEEE Computer Society Press, Los Alamitos, Calif.
12. P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.
13. M. Rinard and M. Lam. The design, implementation, and evaluation of jade. *ACM Transactions on Programming Languages and Systems*, 20(3):483–545, May 1998.
14. R. Rugina and M. Rinard. Automatic parallelization of divide and conquer algorithms. In *Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Atlanta, GA, May 1999.
15. R. Rugina and M. Rinard. Pointer analysis for multithreaded programs. In *Proceedings of the SIGPLAN '99 Conference on Program Language Design and Implementation*, Atlanta, GA, May 1999.
16. R. Rugina and M. Rinard. Symbolic bounds analysis of pointers, array indexes, and accessed memory regions. In *Proceedings of the SIGPLAN '00 Conference on Program Language Design and Implementation*, Vancouver, Canada, June 2000.
17. R. Triolet, F. Irigoin, and P. Feautrier. Direct parallelization of CALL statements. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, CA, June 1986.
18. R. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the SIGPLAN '95 Conference on Program Language Design and Implementation*, La Jolla, CA, June 1995.

Software Pipelining of Nested Loops

Kalyan Muthukumar and Gautam Doshi

Intel Corporation

2200 Mission College Blvd., Santa Clara, CA 95052, U.S.A.

{kalyan.muthukumar, gautam.doshi}@intel.com

Abstract. Software pipelining is a technique to improve the performance of a loop by overlapping the execution of several iterations. The execution of a software-pipelined loop goes through three phases: prolog, kernel, and epilog. Software pipelining works best if most of the time is spent in the kernel phase rather than in the prolog or epilog phases. This can happen only if the trip count of a pipelined loop is large enough to amortize the overhead of prolog and epilog phases. When a software-pipelined loop is part of a loop nest, the overhead of filling and draining the pipeline is incurred for every iteration of the outer loop. This paper introduces two novel methods to minimize the overhead of software-pipeline fill/drain in nested loops. In effect, these methods overlap the draining of the software pipeline corresponding to one outer loop iteration with the filling of the software pipeline corresponding to one or more subsequent outer loop iterations. This results in better instruction-level parallelism (ILP) for the loop nest, particularly for loop nests in which the trip counts of inner loops are small. These methods exploit ItaniumTM architecture software pipelining features such as predication, register rotation, and explicit epilog stage control, to minimize the code size overhead associated with such a transformation. However, the key idea behind these methods is applicable to other architectures as well. These methods have been prototyped in the Intel optimizing compiler for the ItaniumTM processor. Experimental results on SPEC2000 benchmark programs are presented.

1 Introduction

Software pipelining [1,2,4,6,7,10,13,14,15,16] is a well known compilation technique that improves the performance of a loop by overlapping the execution of independent instructions from several iterations. The execution of a software-pipelined loop goes through three phases: **prolog**, when the pipeline is filled - i.e. new iterations are commenced and no iterations are completed, **kernel**, when the pipeline is in steady state - i.e. new iterations are commenced and older iterations are completed, and **epilog**, when the pipeline is drained - i.e. no new iterations are commenced and older iterations are completed. See Fig.1(a).

Since maximum instruction-level parallelism (ILP) is obtained during the kernel phase, software pipelining works best if most of the execution time is spent in kernel phase rather than in prolog or epilog phases. This can happen only if

the trip count of the pipelined loop is large enough to amortize the necessary overhead of the prolog and epilog phases. In practice, there are nested loops, in which the outer loop(s) have high trip counts, and the inner loop has a low trip count. In such cases, the inner loop's software pipeline fill/drain overhead is incurred for every outer loop iteration. This overhead is then amortized over only a few inner loop iterations, hence relatively less time is spent in the kernel phase. This results in poor ILP for the loop nest. See Fig.1(b).

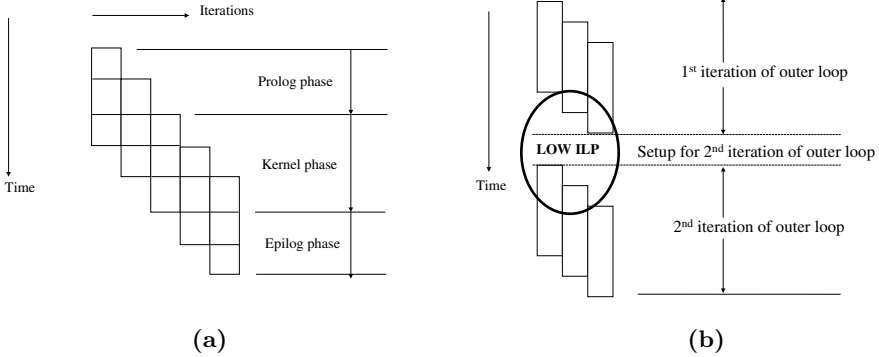


Fig. 1. (a) Phases of a software-pipelined loop, (b) Prolog/Epilog overhead for inner loops with short trip counts.

This paper presents two novel methods to address this problem. These methods perform **Outer Loop Pipelining (OLP)**, by overlapping the epilog of the software pipeline corresponding to one outer loop iteration with the prolog of the software pipelines corresponding to one or more subsequent outer loop iterations. Thus, the software pipeline for the inner loop is filled just once for the loop nest, when the first iterations for all the loops in the loop nest are executed. The software pipeline is also drained just once for the loop nest, when the last iterations for all the loops in the loop nest are executed.

Using ItaniumTM architecture [9, 8] software pipelining features (such as predication [12], register rotation [5], and epilog stage count register), these methods can be implemented with minimal code changes. The inner loop schedule remains unchanged and only a few additional instructions are added to the outer loop. Hence these methods work well even when the trip counts of inner loops are large. In such cases, the performance improvement due to OLP is not as large as when the inner loops have shorter trip counts. Since there is either a small or a large performance gain, and negligible performance penalty for using this technique, it is especially useful for loop nests whose trip counts are not known at compile-time.

These methods have been prototyped in the Intel optimizing compiler for the ItaniumTM processor. Experimental results for kernels derived from workstation applications indicate good speedups for loop nests that have short trip count inner loops. Results on SPECfp2000 and SPECint2000 suites of benchmarks also validate the applicability of this technique for a number of key loops.

The key idea behind these methods can also be applied to architectures that do not have support for rotating registers and other features for software pipelining. In such cases, limited overlap can be achieved between the execution of two successive outer loop iterations, at the expense of some increase in code size.

The rest of this paper is organized as follows. Section 2 describes the ItaniumTM architecture features and the software-pipelining schema using these features. Sections 3 presents two OLP methods that use ItaniumTM architecture features to achieve pipelining of nested loops. Section 4 discusses how OLP can be applied to traditional architectures as well. Section 5 presents experimental results of these methods on the SPECint2000 and SPECfp2000 suites of benchmarks. Finally, Section 6 provides a summary and directions for future work.

Background and Terminology: We use the term *source loop* to refer to the original source code loop, and the term *kernel loop* to refer to the code that implements the software-pipelined version of the source code. Iterations of the source loop are called *source iterations* and iterations of the kernel loop are called *kernel iterations*. Instructions corresponding to a single source iteration are executed in *stages*. A single source iteration spans multiple kernel iterations. The number of cycles between the start of successive kernel iterations is called the **Initiation Interval (II)**. Figure 1(a) shows the execution of five source iterations of a software-pipelined loop with three pipeline stages.

2 Software Pipelining in the ItaniumTM Architecture

The ItaniumTM architecture provides many features to aid the compiler in enhancing and exploiting instruction level parallelism (ILP) [9,8]. These include an explicitly parallel (EPIC) instruction set, large register files, register renaming, predication [12], speculation [11], and special support for software pipelining.

The special support for software-pipelined loops includes register rotation, loop branches and loop control registers. Such features were first seen in the Cydrome Cydra-5 [5]. Register rotation provides a renaming mechanism that eliminates the need to unroll loops for the purpose of software renaming of registers. Registers are renamed by adding the register number to the value of a register rename base (RRB) modulo the size of the rotating register file. The RRB is decremented when a software-pipelined loop branch is executed at the end of each kernel iteration. Decrementing the RRB makes the value in register X, during one kernel iteration, appear to move to register X+1, in the next kernel iteration. If X is the highest numbered rotating register, its value wraps to the lowest numbered rotating register. General registers r32-r127, floating-point registers f32-f127, and predicate registers p16-p63 can rotate. Registers r0-r31, f0-r31 and p0-p15 do not rotate and are referred to as *static* registers.

Below is an example of register rotation.

```
L1:      ld4      r32 = [r4],4      // post increment r4 by 4
        add      r34 = r34,r9
        st4      [r5] = r35,4      // post increment r5 by 4
        swp_branch L1 ;;          // software pipeline branch
```

Each stage of the software pipeline is one cycle long ($II = 1$), a load latency of 2 cycles and an add latency of 1 cycle is assumed. The value that the load writes to r32 is read by the add, two kernel iterations (and hence two rotations) later, as r34. In the meantime, two more instances of the load are executed. However, because of register rotation, those instances write to different registers and do not destroy the value needed by the add.

Predication refers to the conditional execution of an instruction based on a boolean source operand called the qualifying predicate. If the qualifying predicate is True (one), the instruction is executed. If the qualifying predicate is False (zero), the instruction generally behaves like a no-op. Predicates are assigned values by compare, test-bit, or software-pipelined loop branch instructions. Compare instructions generally write two complementary destination predicate registers based on the boolean evaluation of the compare condition.

The rotation of predicate registers serves two purposes. The first, similar to the rotating general and floating-point registers, is to avoid overwriting a predicate value that is still needed. The second purpose is to control the filling and draining of the software pipeline. To do the latter, a predicate is assigned to each stage of the software pipeline to control the execution of the instructions in that stage. This predicate is called a stage predicate. For counted loops, p16 is architecturally defined to be the stage predicate for the first stage, p17 is defined to be the stage predicate for the second stage, etc. A register rotation takes place at the end of each stage (when the `swp_branch` is executed in the kernel loop). When p16 is set to 1, it enables the first stage for a given source iteration. This value of p16 is rotated to p17 when the `swp_branch` is executed, to enable the second stage for the same source iteration. Each 1 written into p16, sequentially enables all the stages for a given source iteration. This behavior is used to enable (propagate 1s) or disable (propagate 0s) the execution of the stages of the pipelined loop during the prolog, kernel, and epilog phases.

ItaniumTM architecture provides special software-pipeline loop branches for counted (`br.ctop`, `br.cexit`) and while (`br.wtop`, `br.wexit`) loops and software-pipeline loop control registers that maintain the loop count (LC) and epilog count (EC). During the prolog and kernel phases, a decision to continue kernel loop execution means that a new source iteration is started. For example, for a counted loop, LC (which is > 0) is decremented to update the count of remaining source iterations. EC is not modified. P63 is set to one. Registers are rotated (so now p16 is set to 1) and the branch (`ctop`) is taken so as to continue the kernel loop execution.

Once LC reaches zero, all required source iterations have been started and the epilog phase is entered. During this phase, a decision to continue kernel loop execution means that the software pipeline has not yet been fully drained. P63 is now set to zero because there are no more new source iterations to start and the instructions that correspond to non-existent source iterations must be disabled. EC is decremented to update the count of the remaining stages for the last source iteration. Registers are rotated and the branch is executed so as to continue the kernel loop execution. When EC reaches one, the pipeline has been fully drained, and the branch is executed so as to exit the kernel loop execution.

A pipelined version of the example counted loop, using ItaniumTM architecture software pipelining features, is shown below assuming an II of 1 cycle and a loop count of 2 source iterations:

```

        mov     pr.rot = 0 ;;      // Clear rotating preds (16-63)
        mov     LC = 1             // LC = loop count - 1
        mov     EC = 4             // EC = loop stage count
        cmp.eq  p16,p0 = r0,r0 ;;  // Set p16 = 1
L1: (p16) ld4    r32 = [r4],4       // Stage1:
      (p18) add  r34 = r34,r9       // Stage3:
      (p19) st4  [r5] = r35,4      // Stage4:
        br.ctop L1 ;;

```

Thus the various ItaniumTM architectural features of register rotation, predication and software-pipelined loop branches and registers, enable extremely compact and efficient software-pipelined loop sequences.

3 Software Pipelining of Nested Loops

This section presents two new methods for outer loop pipelining (OLP), when the innermost loop is software pipelined. OLP is achieved by overlapping the epilog phase of the inner loop pipeline corresponding to one outer loop iteration, with the prolog (and possibly epilog) phases of the inner loop pipelines corresponding to subsequent outer loop iterations. In doing so, the cost associated with filling and draining the inner loop pipeline is incurred only once during the execution of the entire loop nest rather than during every outer loop iteration. As a result, the performance of the loop nest is improved, especially for inner loops with short trip counts.

Consider a loop nest with an inner loop that is software-pipelined with 5 stages and has a trip count of 2. Figure 2 illustrates how these methods overlap the inner loop computations across multiple outer loop iterations.

During OLP, the inner loop's software pipeline is not drained after all the inner loop source iterations for a given outer loop iteration have been started. Rather, the pipeline is frozen, and set-up for the next outer loop iteration is done. The draining continues during the prolog phase of the next outer loop iteration. Eventually, when all the loops in the enclosing loop nest are in their last iterations, the inner loop software pipeline is drained. Note that computations can be overlapped across more than two outer loop iterations. This occurs when the inner loop pipeline never reaches kernel phase. When inner loop pipeline stages are more than twice the number of inner loop iterations, overlap is achieved across three outer loop iterations. Figure 2 shows inner loop computations overlapped across three outer loop iterations.

Two key features of the ItaniumTM architecture enable efficient OLP. They are: (1) rotating registers, and (2) explicit epilog stage control. Rotating registers enable holding the intermediate results of the inner loop computations corresponding to one outer loop iteration, and at the same time, start the inner loop computations for another outer loop iteration. Epilog stage control is achieved via the EC register. Normally EC would have been initialized such that

the inner loop is drained completely for each outer loop iteration. In these OLP methods, EC is set to 1 at the start of an inner loop for all outer loop iterations, except for the very last iterations of the outer loops, in which case it is set appropriately so that the inner loop pipeline is completely drained.

The two methods differ in their schemas as to how this is accomplished. The first method does not make copies of the kernel code for the inner loop, but the second method does. The first method works only for counted outer loops, while the second method is more general and does not impose this restriction. Except for the copy of the kernel code in the case of the second method, both methods add very few (static and dynamic) instructions to perform OLP. Since these methods overlap computations across outer loop iterations, they must preserve the register and memory dependences of the original loop nest. This is explained later in this section.

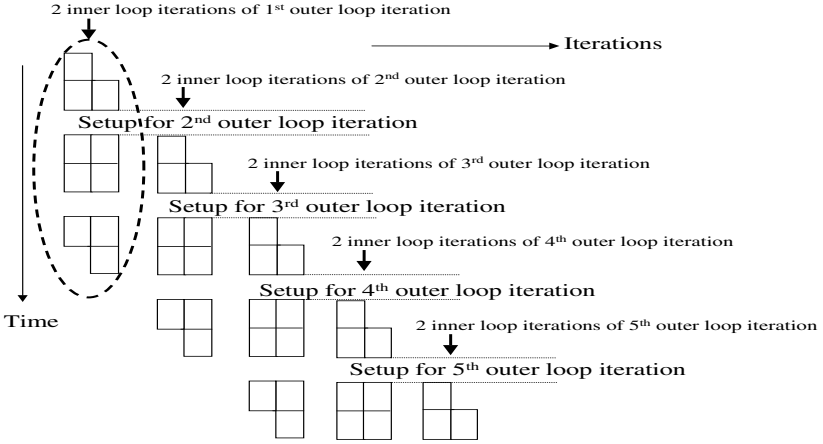


Fig. 2. Overlapping of inner loop pipelines across outer loop iterations.

These methods can be generally applied to perfect or imperfect loop nests of arbitrary depth. For illustration, a running example of a two level perfect loop nest with a counted outer loop is used.

Consider the following loop nest (an excerpt from a critical loop in an important proprietary workstation application), which has a high trip count for the outer loop but a low trip count for the inner loop:

```
REAL*4  A(10,100), B(10,100), C(10,100)
DO  J = 1, 100
  DO  I = 1, 3
    A(I,J) = B(I,J) / C(I,J)
  ENDDO
ENDDO
```

Suppose that the inner loop is pipelined in 12 stages and has an II of 5 cycles¹. Furthermore, suppose that it takes 5 cycles to setup the pipeline (i.e. reset array addresses, reset rotating predicate registers, reset EC/LC, etc). Then, this loop requires roughly:

¹ Divides in ItaniumTM architecture are implemented as a sequence of instructions [9].

```

Cycles for prolog and kernel stages of inner loop = 100*( 3*5)
Cycles for epilog stages of inner loop           = 100*(11*5)
Cycles to reset for the next outer loop iteration = 100*( 5)
-----
TOTAL CYCLES                                     = 7500 cycles

```

The overhead for draining the epilog stages is very high (5500 cycles i.e. 73% of the total cycles). Traditional techniques of loop collapsing, loop interchange and loop unrolling can be used to address this problem but they each have their own costs. Loop collapsing adds predicated address computations that could affect the inner loop scheduled II. Loop interchange could adversely affect (as it does in this example) the memory access pattern. Loop unrolling increases code size and cannot be easily applied if the inner loop trip count is not a compile-time constant.

Without OLP, the generated code is as follows:

```

        mov     r14 = 99          // Outer loop count - 1
        mov     r2 = 1           // Outer loop index
Outer_loop:
        mov     pr.rot = 0       // P16-P63=0
        mov     EC = 12          // EC = Stage count
        mov     LC = 2 ;;        // LC = Trip count - 1
        cmp.eq  p16,p0 = r0,r0   // P16 = 1
        .....
Inner_loop:
        [inner loop code]
        br.ctop Inner_loop      // Inner loop branch
        .....
        cmp.le  p7,p6 = r2,r14   // Test for outer loop count
        add     r2 = r2,1        // Increment outer loop index
(p7)    br.cond Outer_loop      // Outer loop branch

```

Fig. 3. Non OLP code for the Running Example.

The computations in the inner loop (loads of B and C, the FP divide sequence, and the store of A) have been omitted to simplify the example. Since the inner loop is a counted loop, it uses the `br.ctop` instruction. The LC register is initialized to 2 for every iteration of the inner loop, since the inner loop has a trip count of 3 (for a trip count of N, LC is initialized to (N-1)). The EC register is initialized to 12, the stage count of the pipeline.

3.1 Method 1 for Pipelining of Nested Loops

In this method, EC is initialized (to 1) so that the inner loop pipeline is not drained. A test for the final iteration is inserted in the outer loop to conditionally set EC to completely drain the pipeline. The predicate registers (that control the staging of the pipeline) are preserved across outer loop iterations by not clearing them at the start of each inner loop pipeline. The resultant code is as follows:

```

        mov     r14 = 99
        mov     r2 = 1
        mov     pr.rot = 0 ;;           // (1)
        mov     EC = 1                  // (2)
Outer_loop:
        mov     LC = 2
        cmp.eq  p16,p0 = r0,r0
        .....
Inner_loop:
        [inner loop code]
        br.ctop Inner_loop
        .....
        cmp.eq  p8,p9 = r2,r14 ;;       // (3)
(p9)  mov     EC = 1                    // (4)
(p8)  mov     EC = 12                   // (5)
        cmp.le  p7,p0 = r2,r14
        add     r2 = r2,1
(p7)  br.cond  Outer_loop

```

Fig. 4. Code After Pipelining of Nested Loops

Instructions to clear the rotating predicates (1) and to set EC (2), have been moved to the preheader of the loop nest. Also, EC has been initialized to 1 instead of 12 to prevent the draining of the inner loop software pipeline. Since the clearing of the rotating predicates is now done outside the loop nest, the rotating predicates p17-p63 retain the values between outer loop iterations. This enables the freezing and restarting of the inner loop pipeline.

Instruction (3) checks for the start of the last iteration of the outer loop. If this is the case, then EC is set to 12 by instruction (5). Otherwise, instruction (4) resets EC to 1. Thus, the inner loop pipeline is drained only during the last iteration of the outer loop.

Thus this method requires the addition of just three new instructions to the outer loop:

- Instruction (3) to check for the start of the last iteration of outer loop,
- Instruction (5) to set EC to stage count, for the last iteration of outer loop,
- Instruction (4) to reset EC to 1, for all other iterations of outer loop.

Assume that the addition of these instructions increases the time required to set-up for the next outer loop iteration from 5 to 6 cycles. However, the pipeline is drained only once, so the time required to execute the loop nest is:

Cycles for prolog and kernel stages of inner loop	= 100*(3*5)
Cycles for epilog stages of inner loop	= (11*5)
Cycles to reset for the next outer loop iteration	= 100*(6)

TOTAL CYCLES	= 2155 cycles

Thus, this method leads to a significant performance improvement (71%) over that of the original code sequence, with hardly any increase in the static or dynamic code size.

Conditions required for this method: OLP essentially involves hoisting inner-loop code associated with later outer loop iterations, so as to overlap them with the current outer loop iteration. As with all code motion, this hoisting must honor the register and memory data dependences of the original code sequence. This section details the conditions that must be satisfied for this method to work correctly. The following are the key ideas behind these conditions, which ensure the correctness of the OLP transformation:

- We should be able to predict the last iteration(s) of the outer loop(s) and ensure that the pipeline of the inner loop is finally drained.
- Register values that are in flight across inner loop kernel iterations, must not be clobbered due to OLP.
- Live-out register values must be computed correctly, even though the pipeline has not been completely drained.
- All memory-dependences across outer-loop iterations must be obeyed.

Here are the conditions that ensure the correctness of OLP for a loop nest:

1. **Counted Outer Loops:** All the outer loops must be counted loops. This is needed to set EC to drain the software pipeline only when all the outer loops are in their last iterations.
2. **Single Outer Loop Exit:** Each loop in the loop nest should have only one exit. If this is not satisfied, an early exit in one of the loops would transfer control and skip the instructions in the post-exit of the inner loop that set EC to drain the pipeline of the inner loop.
3. **Live-in Values:** If a value is live-in to the inner loop, it should either be in a rotating register or used only in the first stage of the software pipeline. This condition ensures that register and memory anti-dependences across outer loop iterations in the original loop nest are honored.

Figure 5(a) shows a scenario in which R10 is loop-invariant in the inner loop and is used in the second stage, but changes its value in the immediate outer loop. The inner loop has 2 iterations. After OLP, the second iteration of the inner loop still expects to see “value1” in R10. However, this is clobbered with “value2” when the second iteration of the outer loop is started.

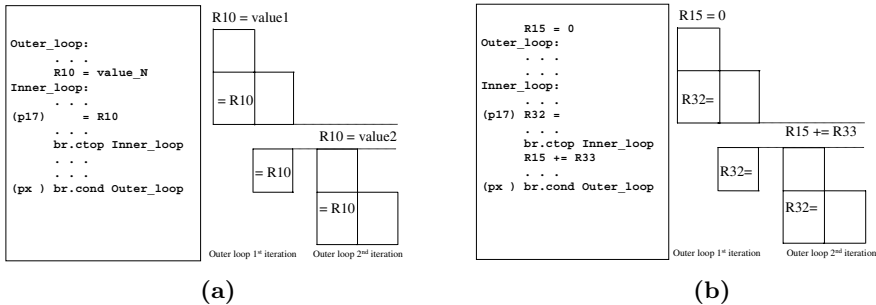


Fig. 5. (a) Register Anti-dependences for live-in values (b) Register dependences for live-out values

Note that this problem will not arise if either (a) the values that are live-in are used in the first stage of the pipeline or (b) the live-in values are assigned to rotating registers. In such cases, the live-in values are not clobbered by their subsequent redefinitions in the outer loops.

4. Live-out Values: If a value is live-out of the inner loop, and is used within the loop nest in a subsequent outer iteration, it should be defined in the first stage of the software pipeline. This condition ensures that register and memory dependences that exist from the inner loop to an outer loop computation are honored. Consider the loop nest in Fig. 5(b), in which the inner loop has 2 stages and a trip count of 2.

The value defined in R32 in the second iteration of the inner loop is live-out of the inner loop. It gets rotated into R33 and is used to update R15. However, if OLP is done, the second stage of the second iteration of the inner loop does not get executed until after the next iteration of the outer loop is started. The result is that the value that is stored in R15 at the start of the second iteration of the outer loop is the value of R32 that is defined in the first iteration of the inner loop. This would obviously be incorrect.

5. Rotating Register Values: Code in the outer loop(s) should not clobber the values held in rotating registers used in the kernel of the inner loop. The register allocator ensures that the rotating registers are not used for outer loop live ranges.

6. Loop-carried Memory Dependence: If there is a loop-carried memory dependence for inner loop computations carried by the outer loop, the Point of First Reference (P_{r1}) of a memory location should precede its Point of Second Reference (P_{r2}) in the execution time-line of these instructions. The memory dependence that exists for the pair (P_{r1} , P_{r2}) can be a *flow*, *anti* or *output* dependence. Depending on the number of inner loop iterations that elapse between P_{r1} and P_{r2} , and the stages in which P_{r1} and P_{r2} occur, it may or may not be legal to perform OLP.

Consider the following:

```
DO J = 1, 100
  DO I = 1, 3
    A(I, J-1) = A(I, J)
  ENDDO
ENDDO
```

For this program, there is a memory flow dependence carried by the outer loop for the location A(I, J). P_{r1} is the definition of the value in A(I,J) and P_{r2} is the subsequent use of that value (referenced as A(I,J-1)). P_{r1} and P_{r2} are defined in terms of number of stages of the software pipeline of the inner loop. In general, let P_{r1} and P_{r2} occur in stages D and U, respectively. If M inner loop iterations elapse between P_{r1} and P_{r2} , then :

$P_{r1} = D$, $P_{r2} = U + M$ and OLP is legal iff $P_{r1} < P_{r2}$ (i.e. iff $D < U + M$).

For the above loop nest, $M = 3$ since three inner loop iterations separate P_{r1} and P_{r2} . If P_{r1} occurs in the 4th stage ($D = 4$) and P_{r2} occurs in the 2nd stage ($U = 2$), then $D < U + M$, so OLP can be done (Fig. 6(a)). However, if $D = 5$ and $U = 1$, then $D > U + M$, and OLP cannot be done (Fig. 6(b)).

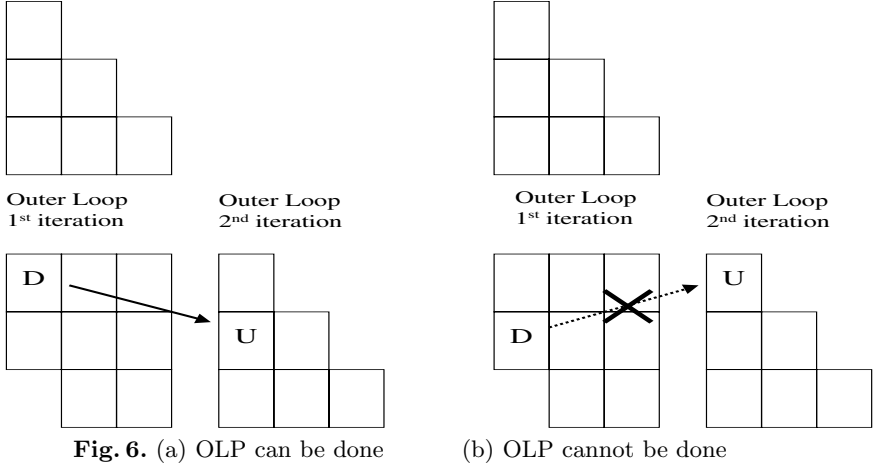


Fig. 6. (a) OLP can be done

(b) OLP cannot be done

The value of M is calculated using (a) the number of outer loop iterations that separate P_{r1} from P_{r2} , which is obtained from data dependence analysis [317], and (b) the number of inner loop iterations that are executed per outer loop iteration. If the number of inner loop iterations is a compile-time constant, then the legality check for loop-carried memory dependence can be performed at compile-time. Otherwise, based on a run-time check of the number of inner loop iterations, control can either go to a loop nest that has OLP or to another loop nest in which only the inner loop is pipelined.

The conditions described above for the live-in and live-out values and the loop-carried memory dependence in the inner loop are applicable only if we do not want any draining of the software pipeline before starting the next outer loop iterations. These conditions can be relaxed if partial draining is allowed to happen at the end of execution of every iteration of the immediate outer loop. This can be formulated as follows:

- Let $S_{live-in}$ be the maximum stage in which a live-in register value that is in a static register is used in the pipelined inner loop.
- Let $S_{live-out}$ be the maximum stage in which a live-out value that is used in the loop nest, is defined in the pipelined inner loop.
- Let $S_{loop-carried-mem-dep}$ denote the maximum of $(P_{r1} - P_{r2})$ over all memory references that have loop-carried dependence across the outer loop(s). Note that the maximum value of $S_{loop-carried-mem-dep}$ is the number of epilog stages of the pipelined inner loop.

So, the value that EC should be set to before starting the next outer loop iteration is: $ES = \text{Max}(1, S_{live-in}, S_{live-out}, S_{loop-carried-mem-dep} + 2)$. If ES is the same as the number of stages required to completely drain the pipeline of the inner loop, then we do not perform OLP, since there is no performance gain from doing it.

Algorithm for Pipelining of Nested Loops: This method can be integrated with modulo scheduling [14] and rotating register allocation in the software

pipeliner. The algorithm for this method consists of two steps. The first step, **IsLoopNestEligibleForOLP**, checks to see if the pipelined inner loop and the loop nest satisfy the conditions listed in the previous section. The second step, **PerformOLPForLoopNest**, performs OLP by suitably adding and moving the instructions so that the pipeline of the inner loop is not completely drained. These two functions are invoked by the main function **OLPForLoopNest**.

Algorithm 1 Algorithm for Method 1

```

Bool IsLoopNestEligibleForOLP (Loop_nest, unsigned int *pES)
{
  if (any outer loop is not a counted loop) return False;
  if (any outer loop has more than one exit) return False;
  Compute  $S_{live-in}$ ,  $S_{live-out}$ , and  $S_{loop-carried-mem-dep}$ ;
  *pES = Max (1,  $S_{live-in}$ ,  $S_{live-out}$ ,  $S_{loop-carried-mem-dep} + 2$ );
  if (*pES == number of pipeline stages) return False;
  else return True;
}

Void PerformOLPForLoopNest (Loop_nest, unsigned int ES)
{
  Move init of pr.rot from preheader of inner loop to preheader of loop nest;
  Delete the initialization of EC in the preheader of the inner loop;
  Initialize EC = ES in the preheader of the loop nest;
  Add a compare instruction in the post-exit of inner loop to set  $p_{Last}$ ;
   $p_{Last} = (\text{last iteration(s) of outer loop(s)}) ? 1 : 0$ ;
  Let  $p_{NotLast}$  = the predicate register that is complementary to  $p_{Last}$ ;
  Add an instruction “( $p_{NotLast}$ ) EC = ES” to post-exit of inner loop;
  Add an instruction “( $p_{Last}$ ) EC = (Epilog Count to completely drain inner
                                loop)” to post-exit of inner loop;
}

Void OLPForLoopNest (Loop_nest)
{
  unsigned int ES;
  if (IsLoopNestEligibleForOLP (Loop_nest, &ES))
    PerformOLPForLoopNest (Loop_nest, ES);
}

```

3.2 Method 2 for Pipelining of Nested Loops

This method is similar in principle to method 1. However, there are three key differences: (a) it does not require that the outer loops in the loop nest be counted loops, (b) it allows outer loops to have multiple exits, and (c) it makes a copy (or copies if there are multiple exits for any outer loop) of the pipelined inner loop and inserts it after the code for the loop nest. Other conditions that were required for method 1 apply to this method as well. This method is conceptually simpler, less restrictive, and allows more loop nests to be pipelined. However,

it comes with the cost of expanded code size and the attendant problems of possibly increased I-cache misses, page faults, instruction TLB misses, etc. This method transforms the code for the running example as follows:

```

        mov      pr.rot = 0 ;;    // (1)
Outer_loop:
        mov      EC = 1          // EC = 1 (no drain)
        mov      LC = 2          // LC = Trip count - 1
        cmp.eq   p16 = r0,r0     // P16 = 1
        ...
Inner_loop:
        [inner loop code]
        br.ctop Inner_loop      // Inner loop branch
        ...
(p7)    br.cond Outer_loop      // Outer loop branch
        mov      EC = 11        // (2)
Inner_loop_copy:
        [inner loop code]
        br.ctop Inner_loop_copy
        <MOV instructions for values that are live-out of Inner_loop_copy>

```

This method consists of the following steps:

- The `pr.rot` instruction that is in the preheader of the inner loop in the non-OLP code is moved out of the loop nest.
- `EC` is now initialized to 1 instead of 12 in the preheader of the inner loop. Immediately following the loop nest, `EC` is set to 11 (number of epilog stages).
- A copy of the pipelined inner loop is placed following this instruction. This serves to drain the inner loop completely.
- Following this copy of the inner loop, `MOV` instructions are added for those rotating register values that are live out of the inner loop. These values are not used inside the loop nest, but outside.

4 Pipelining of Nested Loops for Other Architectures

The key idea behind these methods can be applied for nested loops in other architectures as well. However, the speedup that can be achieved in such architectures is potentially smaller than in architectures that have rotating registers and predication. Without these features, this technique can be implemented as follows: The inner loop is peeled such that the code for the prolog phase of the pipelined inner loop is peeled out of the inner loop and placed in the preheader of the loop nest. The code for the epilog phase of the pipelined inner loop is peeled as well and placed after the code for the inner loop. This is intertwined with a copy of the prolog phase of the inner loop for the next outer loop iteration.

This method does achieve overlap between the epilog phase of the pipelined inner loop for one iteration of the outer loop with the prolog phase of the pipelined inner loop for the next iteration of the outer loop. However, it may be

Algorithm 2 Algorithm for Method 2

```

Bool IsLoopNestEligibleForOLP (Loop_nest, unsigned int *pES)
{
  Compute  $S_{live-in}$ ,  $S_{live-out}$ , and  $S_{loop-carried-mem-dep}$ ;
  *pES = Max (1,  $S_{live-in}$ ,  $S_{live-out}$ ,  $S_{loop-carried-mem-dep} + 2$ );
  if (*pES == number of pipeline stages) return False;
  else return True;
}
Void PerformOLPForLoopNest (Loop_nest, unsigned int ES)
{
  Move the init of pr.rot in preheader of inner loop to outside the loop nest;
  Replace the ‘mov EC’ instruction in the preheader of the inner
    loop with a MOV instruction that initializes EC to ES;
  At each exit of the loop nest, append an instruction “EC = (Stage count - ES)”
    and a copy of the pipelined inner loop. ;
  Following this, for all rotating register values that are live-out of the inner
    loop to outside the loop nest, add appropriate MOV instructions;
}

```

very difficult to overlap inner loop computations across multiple outer loop iterations. Also, if the inner loop trip count is fewer than the number of prolog stages, exits out of the prolog phase will further add to the code size and complexity. It leads to increased code size that is caused by copies of prolog and epilog phases of the inner loop. Also, modulo scheduling for traditional architectures requires kernel unrolling and/or MOV instructions that also lead to increased code size. However, it may be still profitable to do OLP using this technique for loop nests that have small trip counts for inner loops.

5 Experimental Results

We have prototyped both the OLP methods in the Intel optimizing compiler for the ItaniumTM processor. The performance of the resulting OLP code was measured on the ItaniumTM processor. The prototype compiler was used to produce code for a critical loop nest (similar to the running example) of an important workstation application. This application was the motivation for developing and implementing this technique in the compiler. This (and similar) kernels showed large (71%) speedups using OLP. Benefits largely accrue from the inner loop trip counts being small and the outer loop trip counts being large. Next, the applicability of this technique was validated using the SPECfp2000 and SPECint2000 benchmark suites (see Table 1). The first column lists the 14 SPECfp2000 benchmarks followed by 12 SPECint2000 benchmarks. The second column shows the total number of loops in each benchmark that are pipelined. This includes loops that are singly nested as well as those that are in loop nests. The third column in the table shows the number of pipelined loops that are in loop nests - these are the loop nests that are candidates for OLP. The fourth column shows

Table 1. Results of Method 2 on SPECfp2000 and SPECint2000

Benchmark	# of pipelined innermost loops	# of pipelined loops within loop nests	# of pipelined loops with sibling loops	# of OLP loop nests	Code size increase due to OLP
168.wupwise	7	0	0	0	0.0%
171.swim	23	20	15	4	1.5%
172.mgrid	12	10	2	6	0.3%
173.applu	38	38	27	7	0.0%
177.mesa	175	75	52	12	0.4%
178.galgel	258	180	52	50	1.7%
179.art	23	21	16	2	0.8%
183.quake	14	11	7	1	0.0%
187.facerec	59	47	5	38	0.6%
188.ammpp	93	38	24	5	2.9%
189.lucas	22	14	2	3	2.7%
191.fma3d	128	50	44	2	0.1%
200.sixtrack	302	262	216	12	0.0%
301.apsi	132	83	40	35	0.8%
164.gzip	52	21	17	0	0.0%
175.vpr	53	29	15	8	1.1%
176.gcc	205	58	31	1	0.0%
181.mcf	20	6	5	0	0.0%
186.crafty	39	14	7	2	0.0%
197.parser	47	21	15	0	0.0%
252.eon	94	72	4	64	0.4%
253.perlbmk	81	44	22	0	0.0%
254.gap	302	102	76	1	0.0%
255.vortex	17	5	1	0	0.0%
256.bzip2	32	13	11	0	0.0%
300.twolf	223	113	72	9	0.8%

the number of pipelined inner loops that have sibling loops in loop nests. Such loops are not candidates for OLP. The fifth column shows the number of loop nests for which OLP was successfully done using Method 2. Unfortunately, the performance gains due to OLP on these benchmarks were negligible, because: (a) Many critical loop nests have large trip counts for the innermost loop. In such cases, the draining of the software pipeline for the innermost loop is not a high overhead, and therefore reducing the cost of draining the pipeline does not contribute to a significant performance gain. (b) Many critical pipelined loops were ineligible for OLP since they had sibling loops in their loop nests. (c) Live values between stages of the software pipelined loop are exposed by OLP. As a result, the register pressure for the code sections outside the innermost loop increases, causing spills in some cases.

These results validate that the overhead of OLP due to adding instructions in post-exits of inner loops is miniscule (column 6). Even the small performance gain possible in loop nests with large trip counts of inner loops was realized.

6 Conclusion

We have presented two methods for Outer Loop Pipelining (OLP) of loop nests that have software-pipelined inner loops. These methods overlap the draining of the software pipeline corresponding to one outer loop iteration with the filling or draining of the software pipeline corresponding to another outer loop iteration. Thus, the software pipeline for the inner loop is filled and drained only once for the loop nest. This is efficiently implemented using the ItaniumTM architecture features such as predication, rotating registers, and explicit epilog stage control. This technique is applicable, in a limited sense, to other architectures as well.

Both methods are applicable to perfect as well as imperfect loop nests. The first method does OLP with minimal code expansion, but requires all the outer loops be counted loops with no early exits. The second method does not place such restrictions on the loop nest, but does duplicate the kernel code of the pipelined inner loop where required. These methods have been prototyped in the Intel Optimizing Compiler for ItaniumTM architecture. Experimental results indicate good speedups for loop nests with short trip counts for inner loops in an important workstation application. The speedups observed for the SPECfp2000 and SPECint2000 suites of benchmarks were small - this is because their critical loop nests have large trip counts for inner loops.

With the continuing trend of wider and deeply pipelined processors, the availability of parallel execution resources and the latency of instructions will increase. As a result, the prolog/epilog overhead (as a fraction of the execution time of the loop) will increase as well. OLP will be increasingly important as a means to maximize the performance of loop nests.

Acknowledgements. Comments from Dan Lavery, Dong-Yuan Chen, Youfeng Wu, Wei Li, Jean-Francois Collard, Yong-Fong Lee, Sun Chan, and anonymous reviewers helped improve the presentation of this paper. Dan Lavery provided the description of the ItaniumTM architecture software pipelining features. Were it not for the plentiful L^AT_EX support of Kishore Menezes, we'd still be struggling with Word.

References

1. Aiken, A., Nicolau, A.: Optimal Loop Parallelization. Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation, June, (1988), 308–317
2. Allan, Vicki H., Jones, Reese B., Lee, Randall M., Allan, Stephen J. : Software Pipelining. ACM Computing Surveys, **27**, No. 3, September (1995) 367–432
3. Banerjee, U.: Dependence Analysis for Supercomputing. Kluwer Academic Publishers, Boston, MA, (1993)

4. Charlesworth, A.: An Approach to Scientific Array Processing: The Architectural Design of the AP-120B/FPS-164 Family. *IEEE Computer*, Sept. (1981).
5. Dehnert, J. C., Hsu, P. Y., Bratt, J. P.: Overlapped Loop Support in the Cydra 5. *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, April, (1989), 26–38
6. Ebcioğlu, K.: A Compilation Technique for Software Pipelining of Loops with Conditional Jumps. *Proceedings of the 20th Annual Workshop on Microprogramming and Microarchitecture*, Dec. (1987), 69–79
7. Eisenbeis, C., et. al: A New Fast Algorithm for Optimal Register Allocation in Modulo Scheduled Loops. *INRIA TR-RR3337*, January (1998)
8. Huck, J., et al: Introducing the IA-64 Architecture. *IEEE Micro*, **20**, Number 5, Sep/Oct (2000)
9. Intel Corporation: IA-64 Architecture Software Developer's Manual. Santa Clara, CA, April 2000
10. Lam, M. S.: Software Pipelining: An Effective Scheduling Technique for VLIW Machines. *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, June, 1988, 318–328
11. Mahlke, S. A., Chen, W. Y., Hwu, W. W., Rau, B. R., Schlansker, M. S.: Sentinel Scheduling for Superscalar and VLIW Processors. *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct, (1992), 238–247
12. Mahlke, S. A., Hank, R. E., McCormick, J.E., August, D. I., Hwu, W. W.: A Comparison of Full and Partial Predicated Execution Support for ILP Processors. *Proceedings of the 22nd International Symposium on Computer Architecture*, June, (1995), 138–150
13. Rau, B. R., Glaeser, C. D.: Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing. *Proceedings of the 20th Annual Workshop on Microprogramming and Microarchitecture*, Oct, (1981), 183–198
14. Rau, B. R.: Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops. *MICRO-27*, (1994), 63–74
15. Rau, B. R., Schlansker, M. S., Tirumalai, P. P.: Code Generation Schema for Modulo Scheduled Loops. *MICRO-25*, (1992), 158–169
16. Ruttenberg, J., Gao, G. R., Stoutchinin, A., Lichtenstein, W. : Software Pipelining Showdown: Optimal vs. Heuristic Methods in a Production Compiler. *Proceedings of the ACM SIGPLAN 96 Conference on Programming Language Design and Implementation*, May, (1996), 1–11
17. Wolfe, M.: *High-Performance Compilers for Parallel Computing*. Addison-Wesley, Redwood City, CA, (1996)

A First Step Towards Time Optimal Software Pipelining of Loops with Control Flows

Han-Saem Yun¹, Jihong Kim¹, and Soo-Mook Moon²

¹ School of Computer Science and Engineering

² School of Electrical Engineering
Seoul National University, Seoul, Korea

Abstract. We address the problem of time optimal software pipelining of loops with control flows, one of the most difficult open problems in the area of parallelizing compilers. We present a necessary condition for loops with control flows to have equivalent time optimal programs, generalizing the result by Schwiegelshohn *et al.*, which has been the most significant theoretical result on the problem. As part of the formal treatment of the problem, we propose a new formalization of software pipelining, which provides a basis of our proof as well as a new theoretical framework for software pipelining research. Being the *first* generalized result on the problem, our work described in this paper forms an important first step towards time optimal software pipelining.

1 Introduction

Software pipelining is a loop parallelization technique for machines that exploit instruction-level parallelism (ILP) such as superscalar or VLIW processors. It transforms a sequential loop so that new iterations can start before preceding ones finish, thus overlapping the execution of multiple iterations in a pipelined fashion. Since most of the program execution time is spent in loops, much effort has been given to developing various software pipelining techniques.

One of the important theoretical open problems in software pipelining is how to test if a loop with control flows has its equivalent time optimal program or not. A program is time optimal if every execution path p of the program runs in its minimum execution time determined by the length of the longest data dependence chain in p [1]. If decidable, time optimality can be used as a useful measure in evaluating (or improving) existing software pipelining algorithms.

Although there were several related research investigations [2,3,4] on the problem of time optimal software pipelining of loops with control flows, there have been few significant theoretical results published. The work by Schwiegelshohn *et al.* [1] is the best known and most significant result so far, which simply illustrated that, for some loops with control flows, there cannot exist time optimal parallel programs. Their work lacks a *formalism* required to develop generalized results. To the best of our knowledge, since the work by Schwiegelshohn *et al.* was published, no further research results on the problem

has been reported, possibly having been discouraged by the pessimistic result by the Schwiegelshohn *et al.*'s work.

In this paper, we describe a necessary condition for loops with control flows to have equivalent time optimal programs. Our result is the *first* general theoretical result on the problem, and can be considered as a generalization of the Schwiegelshohn *et al.*'s result. In order to prove the necessary condition in a mathematically concrete fashion, we propose a new formalization of software pipelining, which provides a basis of our proof. The proposed formalization of software pipelining has its own significance in that it provides a new theoretical framework for software pipelining research.

We believe that the work described in this paper forms an important first step towards time optimal software pipelining. Although we do not formally prove in the paper, we strongly believe that the necessary condition described in this paper is also the sufficient condition for time optimal programs. Our short-term research goal is to verify this claim so that a given loop with control flows can be classified depending on the existence of time optimal program. Ultimately, our goal is to develop a time optimal software pipelining algorithm for loops that satisfy the condition.

The rest of the paper is organized as follows. In Sect.2, we briefly review prior theoretical work on software pipelining. We explain the machine model assumptions and program representation in Sect.3. Section 4 discusses the dependence model. A formal description of software pipelining is presented in Sect.5 while the proof of a necessary condition is provided in Sect.6. We conclude with a summary and directions for future work in Sect.7.

2 Related Work

For loops *without* control flows, there exist several theoretical results [5,6,7,8,9]. When resource constraints are not present, both the time optimal schedule and the rate optimal one can be found in polynomial time [5,6]. With resource constraints, the problem of finding the optimal schedule is NP-hard in its full generality [6] but there exist approximation algorithms that guarantee the worst case performance of roughly twice the optimum [6,9].

Given sufficient resources, an acyclic program can be always transformed into an equivalent time optimal program by applying list scheduling to each execution path and then simultaneously executing all the execution paths parallelized by list scheduling. When resources are limited, definitions of time optimality may be based on the average execution time. For acyclic programs, Gasperoni and Schwiegelshohn defined an optimality measure based on the execution probability of various execution paths and showed that a generalized list scheduling heuristic guarantees the worst case performance of at most $2 - 1/m + (1 - 1/m) \cdot 1/2 \cdot \lceil \log_2 m \rceil$ times the optimum [10] where m is the number of operations that can be executed concurrently. For loops with control flows, measures based on the execution probability of paths is not feasible, since there are infinitely many execution paths.

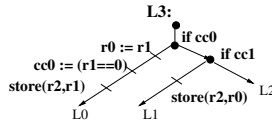


Fig. 1. A tree VLIW instruction

There are few theoretical results for loops with control flows, and, to the best of our knowledge, only two results [11, 12] have been published. The work by Uht [12] proved that the resource requirement necessary for the optimal execution may increase exponentially for some loops with control flows. The Uht’s result, however, is based on an idealized hardware model which is not directly relevant to software pipelining. The work by Schwiegelshohn *et al.* [11], which is the most well-known theoretical result on time optimal programs, showed that there are some loops for which no equivalent time optimal programs exist. Although significant, their contribution lacks any formal treatment of the time optimal software pipelining. For example, they do not formally characterize conditions under which a loop does not have an equivalent time optimal program.

3 Preliminaries

3.1 Architectural Requirements

In order that the time optimality is well defined for loops with control flows, some architectural assumptions are necessary. In this paper, we assume the following architectural features for the target machine model: First, the machine can execute multiple branch operations (i.e., *multiway branching* [12]) as well as data operations concurrently. Second, it has an execution mechanism to commit operations depending on the outcome of branching (i.e., *conditional execution* [13]). The former assumption is needed because if multiple branch operations have to be executed sequentially, time optimal execution cannot be defined. The latter one is also indispensable for time optimal execution, since it enables to avoid output dependence of store operations which belong to different execution paths of a parallel instruction as pointed out by Aiken *et al.* [14].

As a specific example architecture, we use the tree VLIW architecture model [3, 15], which satisfies the architectural requirements described above. In this architecture, a parallel VLIW instruction, called a tree instruction, is represented by a binary decision tree as shown in Fig. 1. A tree instruction can execute simultaneously ALU and memory operations as well as branch operations. The branch unit of the tree VLIW architecture can decide the branch target in a single cycle [12]. An operation is committed only if it lies in the execution path determined by the branch unit [13].

3.2 Program Representation

We represent a sequential program P_s by a control flow graph (CFG) whose nodes are primitive machine operations. If the sequential program P_s is

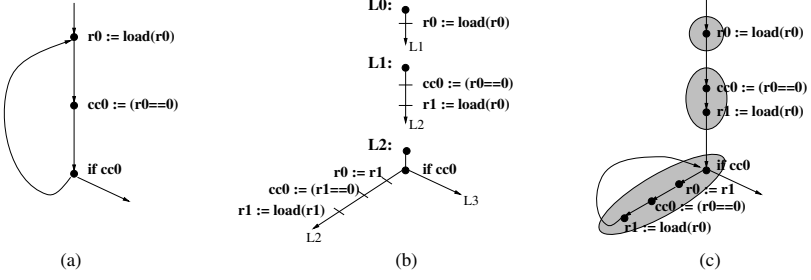


Fig. 2. (a) A sequential program, (b) a parallel tree VLIW program, and (c) a parallel program in the extended sequential representation

parallelized by a compiler, a *parallel tree VLIW program* P_{tree} is generated. While P_{tree} is the final output from the parallelizing compiler for our target architecture, we represent the parallel program in the *extended sequential representation* for the description purpose. Under the extended sequential representation, both sequential programs and parallel programs are described using the same notations and definitions used for the sequential programs. Compared to sequential programs, parallel programs include the additional information on operation grouping. Figure 2 (a) shows an input sequential program P_s and Fig. 2 (b) shows its corresponding parallel tree VLIW program P_{tree} . Using the extended sequential representation, P_{tree} is represented by Fig. 2 (c). The parallel program shown in Fig. 2 (c) is based on a sequential representation except that it has the operation grouping information indicated by shaded regions. The operations belonging to the same group (i.e., the same shaded region) are executed in parallel. A parallel tree VLIW program can be easily converted into the parallel program in the extended sequential representation with some local transformation on copy operations, and vice versa [15].

3.3 Basic Terminology

A program¹ is represented as a triple $\langle G = (N, E), \mathcal{O}, \delta \rangle$. (This representation is due to Aiken *et al.* [14].) The body of the program is a CFG G which consists of a set of nodes N and a set of directed edges E . Nodes in N are categorized into *assignment* nodes that read and write registers or global memory, *branch* nodes that affect the flow of control, and special nodes, *start* and *exit* nodes. The execution begins at the start node and the execution ends at the exit nodes. E represents the possible transitions between the nodes. Except for branch nodes and exit nodes, all the nodes have a single outgoing edge. Each branch node has two outgoing edges while exit nodes have no outgoing edge.

\mathcal{O} is a set of operations that are associated with nodes in N . The operation associated with $n \in N$ is denoted by $op(n)$. More precisely, $op(n)$ represents

¹ Since a parallel program is represented by the extended sequential representation, the notations and definitions explained in Sect. 3.3 and Sect. 4.1 apply to parallel programs as well as sequential programs.

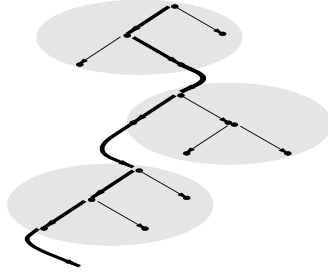


Fig. 3. An execution path in a parallel program

opcode only; Constant fields and register fields are not included in $op(n)$. Without loss of generality, every operation is assumed to write to a single register. We denote by $reg_W(n)$ the register to which n writes and by $regs_R(n)$ a set of registers from which n reads .

A configuration is a pair $\langle n, s \rangle$ where n is a node in N and s is a store (i.e., a snapshot of the contents of registers and memory locations). The transition function δ , which maps configurations into configurations, determines the complete flow of control starting from the initial store. Let n_0 be the start node and s_0 an initial store. Then, the sequence of configurations during an execution is $\langle \langle n_0, s_0 \rangle, \dots, \langle n_i, s_i \rangle, \dots, \langle n_t, s_t \rangle \rangle$ where $\langle n_{i+1}, s_{i+1} \rangle = \delta(\langle n_i, s_i \rangle)$ for $0 \leq i < t$.

A path p of G is a sequence $\langle n_1, \dots, n_k \rangle$ of nodes in N such that $(n_i, n_{i+1}) \in E$ for all $1 \leq i < k$. For a given path p , the number of nodes in p is denoted by $|p|$ and the i -th ($1 \leq i \leq |p|$) node of p is addressed by $p[i]$. A path q is said to be a *subpath* of p , written $q \sqsubseteq p$, if there exists j ($0 \leq j \leq |p| - |q|$) such that $q[i] = p[i + j]$ for all $1 \leq i \leq |q|$. For a path p and i, j ($1 \leq i \leq j \leq |p|$), $p[i, j]$ represents the subpath induced by the sequence of nodes from $p[i]$ up to $p[j]$. Given paths $p_1 = \langle n_1, n_2, \dots, n_k \rangle$ and $p_2 = \langle n_k, n_{k+1}, \dots, n_l \rangle$, $p_1 \circ p_2 = \langle n_1, n_2, \dots, n_k, n_{k+1}, \dots, n_l \rangle$ denotes the concatenated path between p_1 and p_2 . A path p forms a cycle if $p[1] = p[|p|]$ and $|p| > 1$. For a given cycle c , c^k denotes the path constructed by concatenating c with itself k times. Two paths p and q are said to be equivalent, written $p \equiv q$, if $|p| = |q|$ and $p[i] = q[i]$ for all $1 \leq i \leq |p|$.

A path from the start node to one of exit nodes is called an *execution path* and distinguished by the superscript e (e.g., p^e). Each execution path can be represented by an initial store with which the control flows along the execution path. Suppose a program \mathcal{P} is executed with an initial store s_0 and the sequence of configurations is written as $\langle \langle n_0, s_0 \rangle, \langle n_1, s_1 \rangle, \dots, \langle n_f, s_f \rangle \rangle$, where n_0 denotes the start node and n_f one of exit nodes. Then $ep(\mathcal{P}, s_0)$ is defined to be the execution path $\langle n_0, n_1, \dots, n_f \rangle$. (*ep* stands for *execution path*.) Compilers commonly performs the static analysis under the assumption that all the execution paths of the program are executable, because it is undecidable to check if an arbitrary path of the program is executable. In this paper, we make the same assumption, That is, we assume $\forall p^e$ in $\mathcal{P}, \exists s$ such that $p^e \equiv ep(\mathcal{P}, s)$.

It may incur some confusion to define execution paths for a parallel program because the execution of the parallel program consists of transitions among parallel instructions each of which consists of several nodes. With the conditional execution mechanism described in Sect. 3.1, however, we can focus on the unique committed path of each parallel instruction while pruning uncommitted paths. Then, like a sequential program, the execution of a parallel program flows along a single thread of control and corresponds to a path rather than a tree. For example, in Fig. 3, the execution path of a parallel program is distinguished by a thick line.

Some attributes such as redundancy and dependence should be defined in a flow-sensitive manner because they are affected by control flows. Flow-sensitive information can be represented by associating the past and the future control flow with each node. Given a node n and paths p_1 and p_2 , the triple $\langle n, p_1, p_2 \rangle$ is called a *node instance* if $n = p_1[p_1] = p_2[1]$. That is, a node instance $\langle n, p_1, p_2 \rangle$ defines the execution context in which n appears in $p_1 \circ p_2$. In order to distinguish the node instance from the node itself, we use a boldface symbol like \mathbf{n} for the former. The node component of a node instance \mathbf{n} is addressed by $node(\mathbf{n})$. A trace of a path p , written $t(p)$, is a sequence $\langle \mathbf{n}_1, \mathbf{n}_2, \dots, \mathbf{n}_{|p|} \rangle$ of node instances such that $\mathbf{n}_i = \langle p[i], p[1, i], p[i, |p|] \rangle$ for all $1 \leq i \leq |p|$. The i -th component of $t(p)$ is addressed by $t(p)[i]$ and the index of a node instance \mathbf{n} in the trace $t(p)$ is represented by $pos(\mathbf{n})$. For the i -th node instance \mathbf{n}_i of $t(p)$ whose node component is a branch node, a boolean-valued attribute *dir* is defined as follows:

$$dir(\mathbf{n}_i) = \begin{cases} T & \text{if } p[i+1] \text{ is the T-target successor of } p[i] , \\ F & \text{otherwise .} \end{cases}$$

Some of node instances in parallel programs are actually used to affect the control flow or the final store while the others are not. The former ones are said to be *effective* and the latter ones *redundant*. A node is said to be *non-speculative* if all of its node instances are effective. Otherwise it is said to be *speculative*. These terms are further clarified in Sect. 5.

4 Dependence Model

Let alone irregular memory dependences, existing dependence analysis techniques cannot model true dependences accurately mainly because true dependences are detected by conservative analysis on the closed form of programs. In Sect. 4.1 we introduce a path-sensitive dependence model to represent precise dependence information. In order that the schedule is constrained by true dependences only, a compiler should overcome false dependences. We explain how to handle the false dependences in Sect. 4.2.

4.1 True Dependences

With the sound assumption of regular memory dependences, true dependence information can be easily represented for straight line loops thanks to the periodicity of dependence patterns. For loops with control flows, however, this is

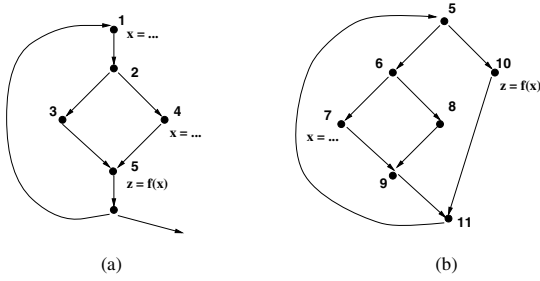


Fig. 4. Path-sensitive dependence examples

not the case and the dependence relationship between two nodes relies on the control flow between them as shown in Fig. 4. In Fig. 4 (a), there are two paths, $p_1 = \langle 1, 2, 3, 5 \rangle$ and $p_2 = \langle 1, 2, 4, 5 \rangle$, from node 1 to node 5. Node 5 is dependent on node 1 along p_1 , but not along p_2 . This ambiguity cannot be resolved unless node 1 is splitted into distinct nodes to be placed in each path. In Fig. 4 (b), node 7 is first used after k iterations of c_1 along $p_3 \circ c_1^k \circ p_4$, where $p_3 = \langle 7, 9, 11 \rangle$, $p_4 = \langle 5, 10 \rangle$ and $c_1 = \langle 5, 6, 8, 9, 11, 5 \rangle$. However, this unspecified number of iterations, k , cannot be modeled by existing techniques; That is, existing techniques cannot model the unspecified dependence distance. In order to model this type of dependence, it is necessary to define the dependence relation on node instances rather than on nodes themselves. The dependences between node instances carried by registers are defined as follows.

Definition 1. Given a path p and i, j ($1 \leq i < j \leq |p|$), $t(p^e)[j]$ is said to be dependent on $t(p^e)[i]$, written $t(p^e)[i] \prec t(p^e)[j]$, if

$$\begin{aligned} \text{reg}_W(p^e[i]) &\in \text{regs}_R(p^e[j]) \text{ and} \\ \text{reg}_W(p^e[k]) &\neq \text{reg}_W(p^e[i]) \text{ for all } i < k < j. \end{aligned}$$

The relation \prec models *true* dependence along p^e , which corresponds to actual definition and uses of values during execution. The relation \prec precisely captures the flow of values through an execution of a program. From Definition 1, we can easily verify the following property on the relation \prec : For execution paths p_1^e and p_2^e , where $1 \leq i_1 < j_1 \leq |p_1^e|$ and $1 \leq i_2 < j_2 \leq |p_2^e|$,

$$\begin{aligned} j_1 - i_1 &= j_2 - i_2 \wedge p_1^e[i_1 + k] = p_2^e[i_2 + k] \text{ for all } 0 \leq k \leq j_1 - i_1 \\ \implies t(p_1^e)[i_1] &\prec t(p_1^e)[j_1] \text{ iff } t(p_2^e)[i_2] \prec t(p_2^e)[j_2]. \end{aligned}$$

The dependence relation between two node instances with memory operations may be irregular even for straight line loops. Existing software pipelining techniques rely on conservative dependence analysis techniques, in which the dependence relationship between two node instances is determined by considering the iteration difference only [16] and is usually represented by *data dependence graphs* [17] or its extensions [18, 19]. The above property holds for

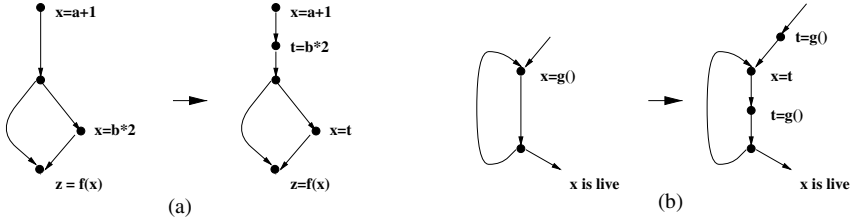


Fig. 5. Copy operations used to overcome false dependences

these representation techniques. In our work, we assume a similar memory dependence relation, in which the dependence relation between node instances with memory operations also satisfies the above property.

4.2 False Dependences

For loops with control flows, it is not a trivial matter to handle false dependences. They cannot be eliminated completely even if each live range is renamed before scheduling. For example, the scheduling techniques described in [14,15] rely on the “on the fly” register allocation scheme based on copy operations so that the schedule is constrained by true dependences only.

In Fig. 5 (a), for the $x = b*2$ to be scheduled above the branch node, x should not be used for the target register of $x = b*2$ and, therefore, the live range from $x = b*2$ to $z = f(x)$ should be renamed. But the live range from $x = b*2$ to $z = f(x)$ alone cannot be renamed because the live range from $x = a+1$ to $z = f(x)$ is combined with the former by x . Thus, the live range is splitted by the copy operation $x = t$ so that t carries the result of $b*2$ along the prohibited region and t passes $b*2$ to x the result.

In Fig. 5 (b), $x = g()$ is to be scheduled across the exit branch but $x = g()$ is used at the exit. So the live range from $x = g()$ to exit is expected to be longer than an iteration, but it cannot be realized if only one register is allocated for the live range due to the register overwrite problem. This can be handled by splitting the long live range into ones each of which does not span more than an iteration, say one from $t = g()$ to $x = t$ and one from $x = t$ to the exit.

In the next section, these copy operations used for renaming are distinguished from ones in the input programs which are byproduct of other optimizations such as common subexpression elimination. The true dependence carried by the live range joined by these copy operations is represented by \prec^* relation as follows.

Definition 2. Given an execution path of a parallel program p^e , let \mathbf{N}_{p^e} represent the set of all node instances in $t(p^e)$. For node instances \mathbf{n} in $t(p^{e,sp})$, $Prop(\mathbf{n})$ represents the set of copy node instances in $t(p^e)$ by which the value defined by \mathbf{n} is propagated, that is,

$$Prop(\mathbf{n}) = \{\mathbf{n}^c \mid \mathbf{n} < \mathbf{n}_1^c, \mathbf{n}_k^c < \mathbf{n}^c, \mathbf{n}_i^c < \mathbf{n}_{i+1}^c \text{ for all } 1 \leq i < k \\ \text{where } \mathbf{n}^c \text{ and } \mathbf{n}_i^c (1 \leq i \leq k) \text{ are copy node instances}\}.$$

For node instances \mathbf{n}_1 and \mathbf{n}_2 in \mathbf{N}_{p^e} , we write $\mathbf{n}_1 \prec^* \mathbf{n}_2$ if

$$\mathbf{n}_1 \prec \mathbf{n}_2 \text{ or } \exists \mathbf{n}^c \in \text{Prop}(\mathbf{n}_1), \mathbf{n}^c \prec \mathbf{n}_2 .$$

Definition 3. The extended live range of \mathbf{n} , written $\text{elr}(\mathbf{n})$, is the union of the live range of the node instance \mathbf{n} and those of copy node instances in $\text{Prop}(\mathbf{n})$, that is,

$$\text{elr}(\mathbf{n}) = t(p)[\text{pos}(\mathbf{n}), \max\{\text{pos}(\mathbf{n}^c) | \mathbf{n}^c \in \text{Prop}(\mathbf{n})\}] .$$

Now we are ready to define a *dependence chain* for sequential and the parallel programs.

Definition 4. Given a path p , a dependence chain d in p is a sequence of node instances in $t(p)$ $\langle \mathbf{n}_1, \mathbf{n}_2, \dots, \mathbf{n}_k \rangle$ such that $\mathbf{n}_i \prec^* \mathbf{n}_{i+1}$ for all $1 \leq i < k$.

The i -th component of a dependence chain d is addressed by $d[i]$ and the number of components in d is denoted by $|d|$.

5 Requirements of Software Pipelining

In this section, we develop a formal account of transformations of software pipelining, which will provide a basis for the proof in Sect. 6. Given an input loop \mathcal{L} and its parallel version \mathcal{L}^{SP} , let \mathbf{P}^e and $\mathbf{P}^{e,\text{SP}}$ denote the set of all execution paths in \mathcal{L} and the set of those in \mathcal{L}^{SP} , respectively. Let us consider a relation $\mathcal{R} : \mathbf{P}^e \times \mathbf{P}^{e,\text{SP}}$ defined by

$$(p^e, p^{e,\text{SP}}) \in \mathcal{R} \text{ iff } \exists \text{ a store } s, \text{ ep}(\mathcal{L}, s) \equiv p^e \wedge \text{ep}(\mathcal{L}^{\text{SP}}, s) \equiv p^{e,\text{SP}} .$$

In order to formalize software pipelining, we are to restrict transformations (that map p^e into $p^{e,\text{SP}}$) by the following five constraints, Constraints 1-5.

First, transformations should exploit only dependence information, that is, they should have only the effect of reordering nodes. Some optimization techniques (e.g., strength reduction and tree height reduction) may reduce the path length by using other semantic properties of programs (e.g., associativity). However, the scheduler is not responsible for such optimizations. These optimizations are performed before/after the scheduling phase.

Additionally, the scheduler is not responsible for eliminating partially dead operation nodes in p^e , which are not used in p^e but may be used in another execution paths. Partially dead operations may become fully dead by some transformations such as moving branch up and can be eliminated on the fly [15], but we assume that they are not eliminated until a post-pass optimization phase. We require that all operation nodes in p^e , dead or not, be also present in $p^{e,\text{SP}}$. Therefore $p^{e,\text{SP}}$ is required to execute the same operations as p^e in an order compatible with the dependences present in p^e . The path $p^{e,\text{SP}}$, however, may have

additional *speculative nodes*² from other execution paths that do not affect the final store of $p^{e,sp}$ and copy operations used for overcoming false dependences [14,15]. Formally, the first constraint on transformations can be given as follows.

Constraint 1. Let N_1 represent the set of all node instances in $t(p^e)$ and let N_2 represent the set of all effective node instances in $t(p^{e,sp})$. Then, there exists a bijective function f from N_1 to N_2 such that

$$\begin{aligned} \forall \mathbf{n} \in N_1, \quad op(node(\mathbf{n})) &= op(node(f(\mathbf{n}))) \quad \text{and} \\ \forall \mathbf{n}, \mathbf{n}' \in N_1, \quad \mathbf{n} \prec \mathbf{n}' &\text{ iff } f(\mathbf{n}) \prec^* f(\mathbf{n}') . \end{aligned}$$

In this case, $f(\mathbf{n})$ is said to correspond to \mathbf{n} and we use $sp_ni_{p^e, p^{e,sp}}$ to represent the function f for a pair of such execution paths p^e and $p^{e,sp}$.

Second, the final store³ of $p^{e,sp}$ should be equal to that of p^e to preserve the semantic of \mathcal{L} . For this, we require that for any node $n = node(\mathbf{n})$, where \mathbf{n} is a node instance in $t(p^e)$, if the target register of n is live at the exit of p^e , the value defined by $node(sp_ni_{p^e, p^{e,sp}}(\mathbf{n}))$ should be eventually committed to $reg_W(n)$ along $p^{e,sp}$. For simplicity, we assume that all registers in p^e are regarded as being live at the exit of p^e during software pipelining. The liveness of each node in $p^{e,sp}$ are checked at post-pass dead code elimination optimization phase. Constraint 2 concisely states this condition.

Constraint 2. For any assignment node instance \mathbf{n} in $t(p^e)$ such that $\forall i > pos(\mathbf{n}), reg_W(p^e[i]) \neq reg_W(node(\mathbf{n}))$,

$$\begin{aligned} reg_W(node(\mathbf{n})) &= reg_W(node(sp_ni_{p^e, p^{e,sp}}(\mathbf{n}))) \quad \text{or} \\ reg_W(node(\mathbf{n})) &= reg_W(node(\mathbf{n}^c)) \quad \text{for some node instance } \mathbf{n}^c \in Prop(sp_ni_{p^e, p^{e,sp}}(\mathbf{n})). \end{aligned}$$

It is needed to impose a restriction on registers allocated for speculative nodes. Registers defined by speculative nodes are required to be temporary registers that are not used in \mathcal{L} so as not to affect the final store.

Constraint 3. Let \mathbf{R} be the set of registers that are defined by nodes in \mathcal{L} . Then the target register of each speculative node in \mathcal{L}^{SP} is not contained in \mathbf{R} .

Now we are to impose a restriction to preserve the semantic of branches. Let us consider a branch node instance $\mathbf{n} = t(p^e)[i]$ and the corresponding node instance $\mathbf{n}' = t(p^{e,sp})[i'] = sp_ni_{p^e, p^{e,sp}}(\mathbf{n})$. The role of \mathbf{n} is to separate p^e from the set of execution paths that can be represented by $p^e[1, i] \circ p_f$ where p_f represents any path such that $p_f[1] = p^e[i], p_f[2] \neq p^e[i+1]$ and $p_f[p_f]$ is an exit node in \mathcal{L} . \mathbf{n}' is required to do the same role as \mathbf{n} , that is, it should separate $p^{e,sp}$ from the set of corresponding execution paths. But some of them might already be

² In fact, most complications of the nonexistence proof in Sect. 6 as well as the formalization of software pipelining are due to expanded solution space opened up by branch reordering transformation.

³ Temporary registers are excluded.

separated from $p^{e,sp}$ earlier than \mathbf{n}' due to another speculative branch node, the instance of which in $p^{e,sp}$ is redundant, scheduled above \mathbf{n}' . This constraint can be written as follows.

Constraint 4. *Given an execution path p^e and q^e in \mathcal{L} such that*

$$q^e[1, i] \equiv p^e[1, i] \wedge \text{dir}(t(q^e)[i]) \neq \text{dir}(t(p^e)[i]) ,$$

for any execution path $p^{e,sp}$ and $q^{e,sp}$ such that $(p^e, p^{e,sp})(q^e, q^{e,sp}) \in \mathcal{R}$, there exists a branch node $p^{e,sp}[j]$ ($j \leq i'$) such that

$$q^{e,sp}[1, j] \equiv p^{e,sp}[1, j] \wedge \text{dir}(t(q^{e,sp})[j]) \neq \text{dir}(t(p^{e,sp})[j])$$

where i' is an integer such that $t(p^{e,sp})[i'] = \text{sp_ni}_{p^e, p^{e,sp}}(t(p^e)[i])$.

$p^{e,sp}$ is said to be equivalent to p^e , written $p^e \equiv_{SA} p^{e,sp}$, if Constraints [1-4] are all satisfied. (The subscript *SA* is adapted from the expression “semantically and algorithmically equivalent” in [1].) Constraint [4] can be used to rule out a pathological case, *unification of execution paths*. Two distinct execution paths $p_1^e = ep(\mathcal{L}, s_1)$ and $p_2^e = ep(\mathcal{L}, s_2)$ in \mathcal{L} are said to be *unified* if $ep(\mathcal{L}^{SP}, s_1) \equiv ep(\mathcal{L}^{SP}, s_2)$. Suppose p_1^e is separated from p_2^e by a branch, then $ep(\mathcal{L}^{SP}, s_1)$ must be separated from $ep(\mathcal{L}^{SP}, s_2)$ by some branch by Constraint [4]. So p_1^e and p_2^e cannot be unified.

Let us consider the mapping cardinality of \mathcal{R} . Since distinct execution paths cannot be unified, there is the unique p^e which is related to each $p^{e,sp}$. But there may exist several $p^{e,sp}$'s that are related to the same p^e due to speculative branches. Thus, \mathcal{R} is a one-to-many relation, and if branch nodes are not allowed to be reordered, \mathcal{R} becomes a one-to-one relation. In addition, the domain and image of \mathcal{R} cover the entire \mathbf{P}^e and $\mathbf{P}^{e,sp}$, respectively. Because of our assumption in Sect. 3.3 that all the execution paths are executable, $\forall p^e \in \mathbf{P}^e, \exists s, p^e \equiv ep(\mathcal{L}, s)$ and the domain of \mathcal{R} covers the entire \mathbf{P}^e . When an execution path $p^e \in \mathbf{P}^e$ is splitted into two execution paths $p_1^{e,sp}, p_2^{e,sp} \in \mathbf{P}^{e,sp}$ by scheduling some branch speculatively, it is reasonable for a compiler to assume that these two paths are all executable under the same assumption and that the image of \mathcal{R} cover the entire $\mathbf{P}^{e,sp}$. To be short, \mathcal{R}^{-1} is a surjective function from $\mathbf{P}^{e,sp}$ to \mathbf{P}^e .

Let \mathbf{N} and \mathbf{N}^{SP} represent the set of all node instances in all execution paths in \mathcal{L} and the set of all effective node instances in all execution paths in \mathcal{L}^{SP} , respectively. The following constraint can be derived from the above explanation.

Constraint 5. *There exists a surjective function $\alpha : \mathbf{P}^{e,sp} \Rightarrow \mathbf{P}^e$ such that*

$$\forall p^{e,sp} \in \mathbf{P}^{e,sp}, \alpha(p^{e,sp}) \equiv_{SA} p^{e,sp} .$$

Using α defined in Constraint [5] above and $\text{sp_ni}_{p^e, p^{e,sp}}$ defined in Constraint [1], another useful function β is defined, which maps each node instance in \mathbf{N}^{SP} to its corresponding node instance in \mathbf{N} .

Definition 5. $\beta : \mathbf{N}^{\text{SP}} \Rightarrow \mathbf{N}$ is a surjective function such that

$$\beta(\mathbf{n}^{sp}) = sp_ni_{\alpha(p^{e,sp}), p^{e,sp}}^{-1}(\mathbf{n}^{sp})$$

where $p^{e,sp} \in \mathbf{P}^{e,sp}$ is the unique execution path that contains \mathbf{n}^{sp} .

To the best of our knowledge, all the software pipelining techniques reported in literature satisfy Constraints [15](#).

6 Nonexistence of Time Optimal Solution

In this section, we prove a necessary condition for a loop to have an equivalent time optimal parallel program. Before a formal proof, we first define *time optimality*. For each execution path $p^{e,sp} \in \mathbf{P}^{e,sp}$, the execution time of each node instance \mathbf{n} in $t(p^{e,sp})$ can be counted from the grouping information associated with \mathcal{L}^{SP} and is denoted by $\tau(\mathbf{n})$. Time optimality of the parallel program \mathcal{L}^{SP} is defined as follows [114](#).

Definition 6. (Time Optimality)

\mathcal{L}^{SP} is time optimal, if for every execution path $p^{e,sp} \in \mathbf{P}^{e,sp}$, $\tau(t(p^{e,sp})[p^{e,sp}])$ is the length of the longest dependence chain in the execution path p^e .

The definition is equivalent to saying that every execution path in \mathcal{L}^{SP} runs in the shortest possible time subject to the true dependences. Note that the longest dependence chain in p^e is used instead of that in $p^{e,sp}$ because the latter may contain speculative nodes which should not be considered for the definition of time optimality. Throughout the renaming of the paper, the length of the longest dependence chain in a path p is denoted by $\|p\|$.

For time optimal programs, there have been no significant theoretical results reported, since Schwiegelshohn *et al.* showed that no time optimal parallel programs exist for some loops with control flows. [11](#). In this section, we prove a *strong* necessary condition for a loop to have the equivalent time optimal parallel program. Our work is the *first* theoretical result on time optimality of loops with control flows.

The necessary condition for \mathcal{L} to have its equivalent time optimal parallel program is as follows.

Condition 1. *There exists a constant $B > 0$ such that for any execution path p^e in \mathcal{L}*

$$\|p^e[1, i]\| + \|p^e[j, |p^e|]\| \leq \|p^e\| + B \text{ for all } 1 \leq i < j \leq |p^e|.$$

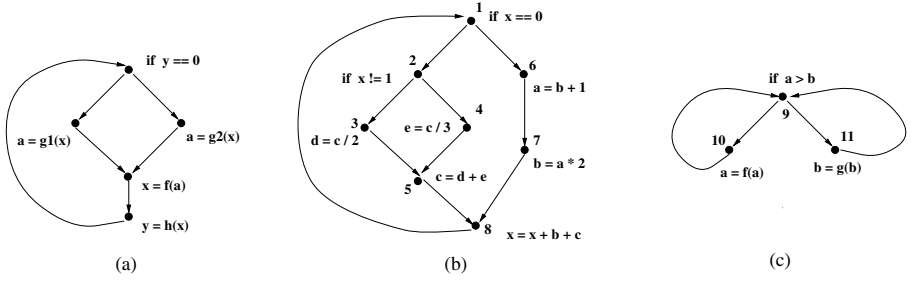


Fig. 6. Example loops used in [1] by Schwiegelshohn *et al.*

Let us consider the example loops shown in Fig. 6. These loops were adapted from [1]. The first one (Fig. 6 (a)), which was shown to have an equivalent time optimal program, satisfies Condition 1. For any execution path p^e that loops k iterations, $\|p^e\| = 2k + 1$ and for $1 \leq i < j \leq |p^e| = 4k$, $\|p^e[1, i]\| \leq \lceil i/2 \rceil + 1$ and $\|p^e[j, |p^e|]\| \leq \lceil 2k - j/2 \rceil + 2$. So,

$$\|p^e[1, i]\| + \|p^e[j, |p^e|]\| \leq 2k + 3 - (j - i)/2 \leq \|p^e\| + 2.$$

The second and third shown in Figs. 6 (b) and 6 (c) do not satisfy Condition 1, thus having no equivalent time optimal programs as shown in [1]. For the loop in Fig. 6 (b), let $c_1 = \langle 1, 2, 4, 5, 8, 1 \rangle$ and $c_2 = \langle 1, 6, 7, 8, 1 \rangle$. For the execution path $p^e(k) = c_1^k \circ c_2^k$, we have :

$$\begin{aligned} \|p^e(k)[1, 5k]\| + \|p^e(k)[5k + 1, |p^e(k)|]\| - \|p^e(k)\| = \\ (2k + 1) + (2k + 1) - (3k + 1) = k + 1. \end{aligned}$$

As k is not bounded, there cannot exist a constant B for the loop and it does not satisfy Condition 1. It can be also shown that the loop in Fig. 6 (c) does not satisfy Condition 1 by a similar way.

Throughout the remaining of this section, we assume that \mathcal{L} does not satisfy Condition 1 and that \mathcal{L}^{SP} is time optimal. Eventually, it is proved that this assumption leads to a contradiction showing that Condition 1 is indeed a necessary condition. Without loss of generality, we assume that every operation takes 1 cycle to execute. An operation that takes k cycles can be transformed into a chaining of k unit-time operations. The following proof is not affected by this transformation.

Lemma 1. *For any $l > 0$, there exists an execution path $p^{e, \text{SP}}$ in \mathcal{L}^{SP} and dependence chains of length l in $p^{e, \text{SP}}$, d_1 and d_2 , which contain only effective node instances such that $\text{pos}(d_1[j]) > \text{pos}(d_2[k])$ and $\text{pos}(\beta(d_1[j])) < \text{pos}(\beta(d_2[k]))$ for any $1 \leq j, k \leq l$.*

Proof. From the assumption that \mathcal{L} does not satisfy Condition 1, there must exist i_1, i_2 ($i_1 < i_2$) and p^e such that $\|p^e[1, i_1]\| + \|p^e[i_2, |p^e|]\| > \|p^e\| + 2 \cdot l$. Note

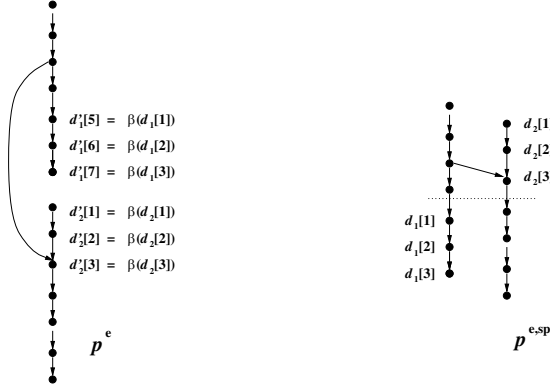


Fig. 7. An example illustrating Lemma 1

that both the terms of LHS is greater than l because otherwise LHS becomes smaller than or equal to $\|p^e\| + l$, a contradiction.

There exist dependence chains d'_1 of length $\|p^e[1, i_1]\|$ and d'_2 of length $\|p^e[i_2, |p^e]|\|$ in p^e such that $\text{pos}(d'_1[\|p^e[1, i_1]\|]) \leq i_1$ and $\text{pos}(d'_2[1]) \geq i_2$. Let $p^{e,sp}$ be an execution path in \mathcal{L}^{SP} such that $\alpha(p^{e,sp}) = p^e$. By Constraint 1, there exist dependence chains d_1 and d_2 of length l in $p^{e,sp}$ such that $\beta(d_1[j]) = d'_1[j - l + \|p^e[1, i_1]\|]$ and $\beta(d_2[k]) = d'_2[k]$ for $1 \leq j, k \leq l$. Then we have for any $1 \leq j, k \leq l$:

$$\text{pos}(\beta(d_1[j])) = \text{pos}(d'_1[j - l + \|p^e[1, i_1]\|]) \leq i_1 < i_2 \leq \text{pos}(d'_2[k]) = \text{pos}(\beta(d_2[k]))$$

Next, consider the ranges for $\tau(d_1[j])$ and $\tau(d_2[k])$, respectively :

$$\begin{aligned} \tau(d_1[j]) &\geq |d'_1[1, j - l + \|p^e[1, i_1]\| - 1]| = j - l + \|p^e[1, i_1]\| - 1 \\ \tau(d_2[k]) &\leq \|p^e\| - |d'_2[k, \|p^e[i_2, |p^e]|\|]| + 1 = \|p^e\| - \|p^e[i_2, |p^e]|\| + k \end{aligned}$$

Consequently, we have for any $1 \leq j, k \leq l$:

$$\tau(d_1[j]) - \tau(d_2[k]) \geq \|p^e[1, i_1]\| + \|p^e[i_2, |p^e]|\| - \|p^e\| + j - k - l + 1 > 0 .$$

Therefore, $\text{pos}(d_1[j]) > \text{pos}(d_2[k])$. □

Figure 7 illustrates Lemma 1 using an example where $l = 3$.

For the rest of this section, we use $p^{e,sp}(l)$ to represent an execution path which satisfies the condition of Lemma 1 for a given $l > 0$, and $d_1(l)$ and $d_2(l)$ are used to represent corresponding d_1 and d_2 , respectively. In addition, let $i_1(l)$ and $i_2(l)$ be i_1 and i_2 , respectively, as used in the proof of Lemma 1 for a given $l > 0$. Finally, $p^e(l)$ represents $\alpha(p^{e,sp}(l))$.

Next, we are to derive the register requirement for “interfering” extended live ranges. $\text{reg}(\text{elr}(\mathbf{n}), \mathbf{n}')$ is used to denote the register which carries $\text{elr}(\mathbf{n})$ at \mathbf{n}' .

Lemma 2. *Given k assignment node instances $\mathbf{n}_1, \mathbf{n}_2, \dots, \mathbf{n}_k$ in an execution path in \mathcal{L}^{SP} and a node instance \mathbf{n} in the execution path, if \mathbf{n} is contained in $\text{elr}(\mathbf{n}_i)$ for all $1 \leq i \leq k$, $\text{reg}(\text{elr}(\mathbf{n}_1), \mathbf{n})$, $\text{reg}(\text{elr}(\mathbf{n}_2), \mathbf{n})$, \dots , $\text{reg}(\text{elr}(\mathbf{n}_k), \mathbf{n})$ are all distinct.*

Proof. The proof is by induction on k . The base case is trivial. For the induction step, assume the above proposition holds for $k = h \geq 1$. Consider $h + 1$ assignment node instances $\mathbf{n}'_1, \mathbf{n}'_2, \dots, \mathbf{n}'_{h+1}$ in an execution path $p^{e, \text{SP}}$ whose extended live ranges share a common node instance \mathbf{n}' . Without loss of generality let us assume $\text{pos}(\mathbf{n}'_{h+1}) > \text{pos}(\mathbf{n}'_i)$ for all $1 \leq i \leq h$. Then the range shared by these extended live ranges can be written as $t(p^{e, \text{SP}})[\text{pos}(\mathbf{n}'_{h+1}), \text{pos}(\mathbf{n}')]]$.

By induction hypothesis, $\text{reg}(\text{elr}(\mathbf{n}'_1), \mathbf{n}'_{h+1}), \dots, \text{reg}(\text{elr}(\mathbf{n}'_h), \mathbf{n}'_{h+1})$ are all distinct. Moreover, $\text{reg}_W(\mathbf{n}'_{h+1})$ must differ from these h registers since the live range defined by \mathbf{n}'_{h+1} interferes with any live ranges carried by these registers. For the same reason at any point in $t(p^{e, \text{SP}})[\text{pos}(\mathbf{n}'_{h+1}), \text{pos}(\mathbf{n}')]]$, any register which carries part of $\text{elr}(\mathbf{n}'_{h+1})$ differs from h distinct registers which carry extended live ranges of \mathbf{n}'_i s. Therefore, the proposition in the above lemma holds for all $k > 0$. \square

For loops without control flows, the live range of a register cannot spans more than an iteration although sometimes it is needed to do so. *Modulo variable expansion* handles this problem by unrolling the software-pipelined loop by sufficiently large times such that II becomes no less than the length of the live range [20]. Techniques based on *Enhanced Pipeline Scheduling* usually overcome this problem by splitting such long live ranges by copy operations during scheduling, which is called as dynamic renaming or partial renaming [15]. Optionally these copy operations are coalesced away after unrolling by a proper number of times to reduce resource pressure burdened by these copy operations. Hardware support such as *rotating register files* simplifies register renaming. For any cases, the longer a live range spans, the more registers or amount of unrolling are needed. There is a similar property for loops with control flows as shown below.

Lemma 3. *Given an effective branch node instance \mathbf{n}_b in an execution path $p^{e, \text{SP}}$ in \mathcal{L}^{SP} and a dependence chain d in $p^{e, \text{SP}}$ such that for any node instance \mathbf{n} in d , $\text{pos}(\mathbf{n}) < \text{pos}(\mathbf{n}_b)$ and $\text{pos}(\beta(\mathbf{n})) > \text{pos}(\beta(\mathbf{n}_b))$, there exist at least $\lfloor |d|/(M + 1) \rfloor - 1$ node instances in d whose extended live ranges contain \mathbf{n}_b where M denotes the length of the longest simple path in \mathcal{L} .*

Proof. Let $p^e = \alpha(p^{e, \text{SP}})$ and $M' = \lfloor |d|/(M + 1) \rfloor$. From the definition of M , there must exist $\text{pos}(\beta(d[1])) \leq i_1 < i_2 < \dots < i_{M'} \leq \text{pos}(\beta(d[|d|]))$ such that $p^e[i_1] = p^e[i_2] = \dots = p^e[i_{M'}]$. If $p^e[i] = p^e[j]$ ($i < j$), there must exist a node instance in p^e , \mathbf{n}' ($i \leq \text{pos}(\mathbf{n}') < j$) such that $\forall k > \text{pos}(\mathbf{n})$, $\text{reg}_W(p^e[k]) \neq \text{reg}_W(\text{node}(\mathbf{n}'))$. Thus by Constraint 2, there must exist node instances in d , $\mathbf{n}_1, \mathbf{n}_2, \dots, \mathbf{n}_{M'-1}$, such that

$$\begin{aligned} \text{reg}_W(\text{node}(\beta(\mathbf{n}_i))) &= \text{reg}_W(\text{node}(\mathbf{n}_i)) \quad \text{or} \\ \text{reg}_W(\text{node}(\beta(\mathbf{n}_i))) &= \text{reg}_W(\text{node}(\mathbf{n}^c)) \\ &\text{for some node instance } \mathbf{n}^c \in \text{Prop}(\mathbf{n}_i) \text{ for all } 1 \leq i \leq M' - 1. \end{aligned}$$

Since $\text{pos}(\mathbf{n}_i) < \text{pos}(\mathbf{n}_b)$ and $\text{pos}(\beta(\mathbf{n}_i)) > \text{pos}(\beta(\mathbf{n}_b))$, $\text{node}(\mathbf{n}_i)$ is speculative for all $1 \leq i \leq M' - 1$. By Constraint 3, $\text{reg}_W(\text{node}(\mathbf{n}_i)) \notin \mathbf{R}$ and the value defined by \mathbf{n}_i cannot be committed into $r \in \mathbf{R}$ until \mathbf{n}_b . So, $\text{elr}(\mathbf{n}_i)$ should contain \mathbf{n}_b for all $1 \leq i \leq M' - 1$. \square

Lemma 4. *Let $\mathbf{N}_b(l)$ represent the set of effective branch node instances in $p^{e,\text{sp}}(l)$ such that for any $\mathbf{n}_b \in \mathbf{N}_b(l)$, $\text{pos}(\beta(\mathbf{n}_b)) \leq i_1(l)$ and $\text{pos}(\mathbf{n}_b) > \text{pos}(d_2(l)[1])$. Then there exists a constant $C > 0$ such that $\tau(\mathbf{n}_b) < \|p^e(l)[1, i_1(l)]\| - 2 \cdot l + C$.*

Proof. Let $C = (M+1)(R+2)$ where M is defined as in Lemma 3 and R denotes the number of registers used in \mathcal{L}^{SP} . Suppose $\tau(\mathbf{n}_b) \geq \|p^e(l)[1, i_1(l)]\| - 2 \cdot l + C$.

From the proof of Lemma 1, $\tau(d_2(l)[C]) \leq \|p^e(l)\| - \|p^e(l)[i_2(l), p^e(l)]\| + C - 1 < \tau(\mathbf{n}_b)$. So at least $\lfloor C/(M+1) \rfloor - 1 = R + 1$ registers are required by Lemmas 2 and 3, a contradiction. So, $\tau(\mathbf{n}_b) < \|p^e(l)[1, i_1(l)]\| - 2 \cdot l + C$. \square

Theorem 1. *Condition 1 is a necessary condition for \mathcal{L} to have an equivalent time optimal program.*

Proof. By Lemma 4, there exist an effective branch node instance \mathbf{n}_b in $p^{e,\text{sp}}(l)$ such that $\tau(\mathbf{n}_b) < \|p^e(l)[1, i_1(l)]\| - 2 \cdot l + C$ and $\tau(\mathbf{n}_b) > \tau(\mathbf{n}'_b)$ where \mathbf{n}'_b represents any branch node instance in $\tau(\mathbf{n}'_b)$ such that $\text{pos}(\beta(\mathbf{n}'_b)) \leq \text{pos}(\beta(d'_1(l)[l]))$.

Let $P(\mathbf{n}_b)$ be the set of execution paths in \mathcal{L}^{SP} such that $q^{e,\text{sp}} \in P(\mathbf{n}_b)$ if $q^{e,\text{sp}}[1, \text{pos}(\mathbf{n}_b)] = p^{e,\text{sp}}(l)[1, \text{pos}(\mathbf{n}_b)]$ and $\text{dir}(t(q^{e,\text{sp}})[\text{pos}(\mathbf{n}_b)]) \neq \text{dir}(t(p^{e,\text{sp}}(l))[\text{pos}(\mathbf{n}_b)])$. Then $\|q^{e,\text{sp}}\| \geq \|p^e(l)[1, i_1(l)]\|$. By Lemma 2, we have $\|q^{e,\text{sp}}[\text{pos}(\mathbf{n}_b) + 1, \|q^{e,\text{sp}}\|]\| > l - C$. Since l is not bounded and C is bounded, the length of any path starting from $\text{node}(\mathbf{n}_b)$ is not bounded, a contradiction. Therefore the assumption that \mathcal{L}^{SP} is time optimal is not valid and Condition 1 is indeed a necessary condition. \square

7 Conclusion

In this paper, we presented a necessary condition for loops with control flows to have their equivalent time optimal programs. The necessary condition described in the paper generalizes the Schwiegelshohn *et al.*'s work, which was lacking for a formalism to produce such a general condition. Based on a newly proposed formalization of software pipelining, we proved the necessary condition in a mathematically concrete fashion.

Our result, which is the first general theoretical result on time optimal software pipelining, is an important first step towards time optimal software pipelining. We strongly believe that the necessary condition presented in the paper is also the sufficient condition. Our immediate future work, therefore, includes the verification of this claim. As a long-term research goal, we plan to develop a time optimal software pipelining algorithm that can generate time optimal programs for eligible loops.

References

1. U. Schwiegelshohn, F. Gasperoni, and K. Ebcioglu. On Optimal Parallelization of Arbitrary Loops. *Journal of Parallel and Distributed Computing*, 11(2):130–134, 1991.
2. A. Aiken and A. Nicolau. Perfect Pipelining. In *Proceedings of the Second European Symposium on Programming*, pages 221–235, June 1988.
3. K. Ebcioglu. A Compilation Technique for Software Pipelining of Loops with Conditional Jumps. In *Proceedings of the 20th Annual Workshop on Microprogramming (Micro-20)*, pages 69–79, 1987.
4. A. Zaky and P. Sadayappan. Optimal Static Scheduling of Sequential Loops with Tests. In *Proceedings of the International Conference on Parallel Processing*, pages 130–137, 1989.
5. A. Aiken and A. Nicolau. Optimal Loop Parallelization. In *Proceedings of the SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 308–317, 1988.
6. F. Gasperoni and U. Schwiegelshohn. Generating Close to Optimum Loop Schedules on Parallel Processors. *Parallel Processing Letters*, 4(4):391–403, 1994.
7. F. Gasperoni and U. Schwiegelshohn. Optimal Loop Scheduling on Multiprocessors: A Pumping Lemma for p-Processor Schedules. In *Proceedings of the 3rd International Conference on Parallel Computing Technologies*, pages 51–56, 1995.
8. L.-F. Chao and E. Sha. Scheduling Data-Flow Graphs via Retiming and Unfolding. *IEEE Transactions on Parallel and Distributed Systems*, 8(12):1259–1267, 1997.
9. P.-Y. Calland, A. Darte, and Y. Robert. Circuit Retiming Applied to Decomposed Software Pipelining. *IEEE Transactions on Parallel and Distributed Systems*, 9(1):24–35, 1998.
10. F. Gasperoni and U. Schwiegelshohn. List Scheduling in the Presence of Branches: A Theoretical Evaluation. *Theoretical Computer Science*, 196(2):347–363, 1998.
11. A. Uht. Requirements for Optimal Execution of Loops with Tests. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):573–581, 1992.
12. S.-M. Moon and S. Carson. Generalized Multi-way Branch Unit for VLIW Microprocessors. *IEEE Transactions on Parallel and Distributed Systems*, pages 850–862, 1995.
13. K. Ebcioglu. Some Design Ideas for a VLIW Architecture for Sequential Natured Software. In *Proceedings of IFIP WG 10.3 Working Conference on Parallel Processing*, pages 3–21, 1988.
14. A. Aiken, A. Nicolau, and S. Novack. Resource-Constrained Software Pipelining. *IEEE Transactions on Parallel and Distributed Systems*, 6(12):1248–1270, 1995.
15. S.-M. Moon and K. Ebcioglu. Parallelizing Non-numerical Code with Selective Scheduling and Software Pipelining. *ACM Transactions on Programming Languages and Systems*, pages 853–898, 1997.

16. V. Allan, R. Jones, R. Lee, and S. Allan. Software Pipelining. *ACM Computing Surveys*, 27(3):367–432, 1995.
17. D. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. Wolfe. Dependence Graphs and Compiler Optimizations. In *SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 207–218, 1981.
18. J. Farrante, K. Ottenstein, and J. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.
19. K. Pingali, M. Beck, R. Johnson, M. Moudgill, and P. Stodghill. Dependence Flow Graphs: An Algebraic Approach to Program Dependences. In *Proceedings of the 1991 Symposium on Principles of Programming Languages*, pages 67–78, 1991.
20. M. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proceedings of the SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 318–328, 1988.

Comparing Tail Duplication with Compensation Code in Single Path Global Instruction Scheduling*

David Gregg

Institut für Computersprachen,
Technische Universität Wien,
Argentinierstr. 8, A-1040 Wien,
E-mail: dave@complang.tuwien.ac.at
Fax: (+431) 58801-18598

Abstract. Global instruction scheduling allows operations to move across basic block boundaries to create tighter schedules. When operations move above control flow joins, some code duplication is generally necessary to preserve semantics. Tail duplication and compensation code are approaches to duplicating the necessary code, used by Superblock Scheduling and Trace Scheduling respectively. Compensation code needs much more engineering effort, but offers the possibility of less code growth. We implemented both algorithms to evaluate whether the extra effort is worthwhile. Experimental results show that trace scheduling does not always create less code growth and often creates more.

1 Introduction

Instruction Level Parallelism (ILP) offers the hope of greatly faster computers by automatically overlapping the execution of many machine-level instructions to complete tasks more quickly. An important class of ILP machine is the Very Long Instruction Word (VLIW) computer. These simple machines provide large numbers of execution resources, but require a sophisticated compiler to schedule the instructions.

A wide variety of scheduling techniques has been proposed, varying from simple basic block scheduling, to sophisticated global software pipelining [Gre00]. The engineering effort in implementing different schemes can vary enormously. A serious problem is that in many cases it is not known whether the benefits of a more sophisticated technique are sufficient to outweigh increased program development and maintenance time. Examples of this include the unclear benefits of DAG scheduling over single path scheduling [Mou94, PSM97], and trade-offs between the very many modulo scheduling algorithms that have been proposed.

Another important example is whether a global instruction scheduling algorithm should use tail duplication or compensation code. Code duplication is

* This work was supported by the Austrian Science Foundation (FWF) Project P13444-INF

often necessary when moving operations across basic block boundaries during scheduling, and a compiler writer can choose between these two approaches. Tail duplication is easy to implement, but duplicates more code than necessary. Compensation code requires much more engineering effort, but offers the possibility of smaller code size and better instruction cache performance.

In this paper, we compare tail duplication [Mah96] and compensation code [Fis81] in the context of single path global instruction scheduling. Single path algorithms schedule the most likely path through a region of code as if it were one large basic block. We focus especially on Superblock Scheduling and Trace Scheduling, global scheduling algorithms that use tail duplication and compensation code respectively. We analyse both approaches, and present experimental results on relative performance in a highly optimising VLIW compiler.

The paper is organised as follows. In section 2 we examine the Trace Scheduling algorithm. Section 3 examines the role of tail duplication in Superblock Scheduling. In section 4 we compare the engineering effort required to implement the two approaches. Section 5 looks at the techniques from the viewpoint of the size of the resulting code. We examine situations where choosing either approach can affect the running time of the compiled program in section 6. Section 7 describes our experimental comparison of the two approaches. In section 8 we examine other work that has compared trade-offs in global scheduling. Finally, we draw conclusions in section 9.

2 Trace Scheduling

We will first describe global instruction scheduling in more detail. The simplest instruction schedulers use local scheduling, and group together independent operations in a single basic block. Global scheduling increases the available instruction level parallelism (ILP) by allowing operations to move across basic block boundaries. One of the earliest global scheduling algorithms is Trace Scheduling [Fis81, FERN84], which optimises the most commonly followed path in an acyclic region of code.

Trace scheduling works in a number of stages. First, traces are identified in the code. Figure 1(a) shows an example of trace selection. A trace is an acyclic sequence of basic blocks which are likely to be executed one after another, based on execution frequency statistics. Traces in an acyclic region are identified starting from the most frequently executed basic block which is not already part of a trace. This block is placed in a new trace, and the algorithm attempts to extend the trace backwards and forwards using the mutual most likely rule. Two blocks a and b , where b is a successor block of a , are mutual most likely if a is most frequently exited through the control flow edge from a to b , and b is most frequently entered through the same edge.

The trace selector continues to extend the trace until there is no mutual most likely block at either end of the trace, or where extending the trace would lead across a loop back-edge, or out of the acyclic region. An important feature of the mutual most likely rule is that at the start or end of any block there will

be at most one other block that satisfies the mutual most likely criteria. This means that each basic block will be in exactly one trace. This simplifies code transformations enormously.

Once the traces have been identified, they are scheduled, with the most frequent traces scheduled first. The scheduler removes the trace from the control flow graph and schedules it as the sequence of basic blocks were a single large basic block. This will re-order the sequence of operations to group together independent operations which can be executed simultaneously. The next step is to place the schedule back into the control flow graph (CFG).

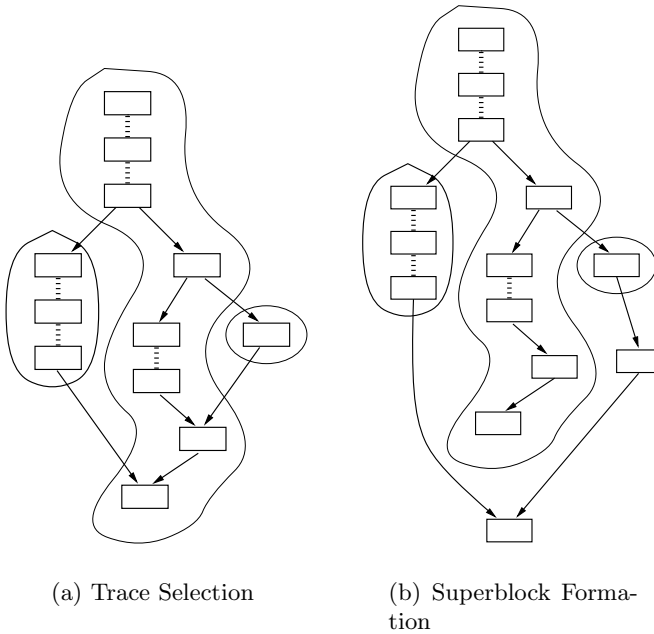
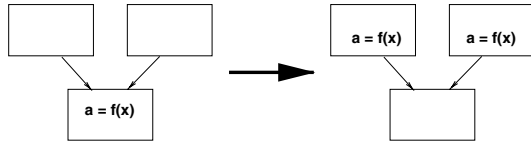


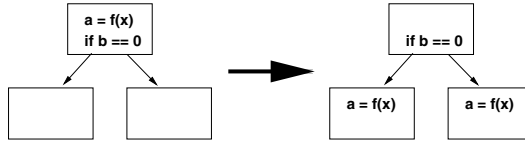
Fig. 1.

Placing the schedule back into the CFG is the most complicated part of Trace Scheduling. The problem is that moving an operation from one basic block to another can break the code. For example, consider the situation in figure 2(a). The operation $a = f(x)$ appears after the control flow join in the original code, but scheduling results in the operation moving above the join. If, for example, the operation were to move into the left-hand block above the join, then if the join were reached during execution via the right-hand block, this operation would never be executed. To overcome this problem, *compensation code* is added at

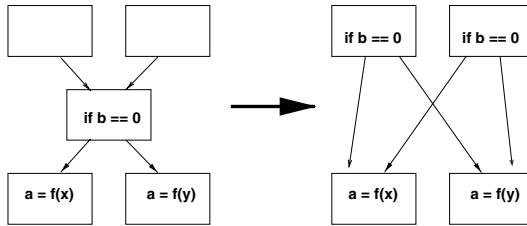
trace entry and exit points to ensure that all operations that should be executed are actually executed.



(a) Moving an operation above a join



(b) Moving an operation below a split



(c) Moving a branch above a join

Fig. 2. Compensation Code

Compensation code is simplest when moving non-branch operations across basic block boundaries. Figure 2(a) shows the effects of moving an operation above a control flow join. The operation must be duplicated and placed on every incoming path to the join. Similarly, figure 2(b) shows how operations may be moved below a control flow split (typically a conditional branch). The operation is copied to all outgoing paths of the split. The situation is considerably more complicated where branches move above joins or below below splits, and often results in significant changes in the shape of the control flow graph. Figure 2(c) shows a simple example of a branch moving above a join.

Generating compensation code is complicated. An operation may move past any combination of splits and/or joins. Merely deciding where to place duplicates

is far from simple. The situation can be even more complicated if renaming or other transformations must be applied at the same time. Modifying the shape of the control flow graph for moving branches across basic block boundaries is also complicated, especially where analysis information in the low level representation must be kept consistent. In order to avoid these engineering difficulties a new approach was taken when building the IMPACT compiler, perhaps the most developed research ILP compiler. In the next section, we describe this simpler approach.

3 Superblock Scheduling

Superblock scheduling [CCMmWH91,MCH⁺92] is similar to trace scheduling in that it attempts to optimise the most frequent path. A superblock is a trace that contains no control flow joins, except at the entry of the trace. Thus, after trace selection the algorithm removes joins from the trace by duplicating blocks which appear after the join. This is known as superblock formation, and an example is shown in figure 1(b). Wherever a block in the middle of a trace has more than one incoming control flow edge, that block is duplicated. It is important to note that duplicating this block will cause its successor blocks to have multiple predecessors, and these blocks must too be duplicated if they are in (but not the first block of) a trace.

After superblock formation, a number of optimisations are applied to each superblock, such as constant propagation, redundant load elimination and unrolling. Removing control flow joins often allows traditional optimisations to be more effective, since there is no need to take account of effects on rarely followed overlapping paths. In addition, Mahlke [Mah96] mentions that the IMPACT compiler also applies branch target expansion to increase the size of superblocks. This is, in effect, another round of tail duplication, as it copies code that appears after joins. In this paper, we do not consider any further duplication after superblock formation.

The next step is to schedule the operations in the superblock. The superblock scheduler is similar to, but simpler than, the scheme used by trace scheduling. The biggest difference is that since superblocks contain no joins, the scheduler does not move operations above joins. This greatly simplifies the code generation process. There is no need to duplicate operations or reshape the control flow graph for branches moving above joins. The superblock scheduler does, however, allow operations to move below branches. Thus, some compensation code may be needed to be added after scheduling. But as described by Mahlke, Superblock Scheduling does not allow branches to be re-ordered, thus avoiding the most complicated part of compensation code for splits.

Superblock formation before scheduling allows almost all of the complexity for compensation code to be eliminated from the compiler. This greatly reduces the engineering effort in implementing and maintaining a global scheduling compiler. In the next section, we look at how great the difference can be.

4 Engineering Effort

There is no good way to compare the engineering difficulty of implementing different code duplication schemes. Any measure will be arbitrary and subjective to a degree. Nonetheless, we believe that some broad conclusions can be drawn. Perhaps the best measure of the complexity of compensation code is that Ellis devotes thirty eight pages of his PhD thesis [Ell85] to describing how to generate it in a Trace Scheduling compiler. The thesis also contains a further fifteen pages of discussion of possible improvements. The treatment is thorough, but not excessive. In contrast, Mahlke adequately describes superblock formation in less than two pages of his PhD thesis [Mah96]. Mahlke does not allow branches to be re-ordered, however, and only briefly describes operations moving below branches. Nonetheless, even if following a similar course would allow half or more of Ellis's material to be removed, his description of compensation code generation would still be very much larger than Mahlke's.

Another possible measure of relative complexity is our own implementation in the Chameleon compiler. Our code to perform tail duplication occupies about 400 lines of source code. It was possible to write this code in less than two weeks. Most of the time was devoted to understanding the Chameleon intermediate representation and debugging. Our code uses existing Chameleon functions for patching up the control flow graph when unrolling loops, however. If these functions were written from scratch, the code might be almost twice as large. In addition, the code is quite simple and straightforward.

For scheduling, Chameleon implements the EPS++ [Mou97] software pipelining algorithm. This algorithm includes quite sophisticated compensation code generation, including compensation code for operations moving across loop iterations. It does not re-order branches, however, but it does include some other optimisations, such as redundancy elimination. After we adapted the EPS++ code for moving operations across basic block boundaries to Trace Scheduling, it occupied more than 4500 lines of code. Not all this code deals with compensation code generation, but even if only half of this code is really necessary for the compensation code in Trace Scheduling, it is still substantially larger than the code for superblock formation. In addition, this code is sometimes quite complicated, especially when moving branches above joins.

5 Code Growth

A simple characterisation of the difference between compensation code and tail duplication is that the former duplicates operations only where necessary, whereas the latter creates code growth whether it is needed or not. In general we would expect that tail duplication would produce substantially more code growth. In fact, this is not necessarily the case.

For an original region containing n operations, the original Trace Scheduling algorithm can create a final schedule containing approximately $O(n^n)$ operations [Ell85]. Most of this worst case code growth comes from reordering branches

which require large amounts of code duplication. A common strategy in global scheduling compilers is to disallow reordering of branches. This eliminates a complicated transformation from the compiler. In addition, the lost ILP from preventing some branches from being scheduled a few cycles earlier is probably low, since most branches are very predictable. This also reduces the worst case code growth from approximately $O(n^n)$ to $O(k^n)$, where k is the largest number of incoming control flow edges for any node in the CFG. The new worst case code growth corresponds to a situation where an acyclic graph is converted to a tree by repeatedly duplicating all nodes that have more than one predecessor. While still exponential, it is an enormous improvement. This is the strategy we follow in our implementation.

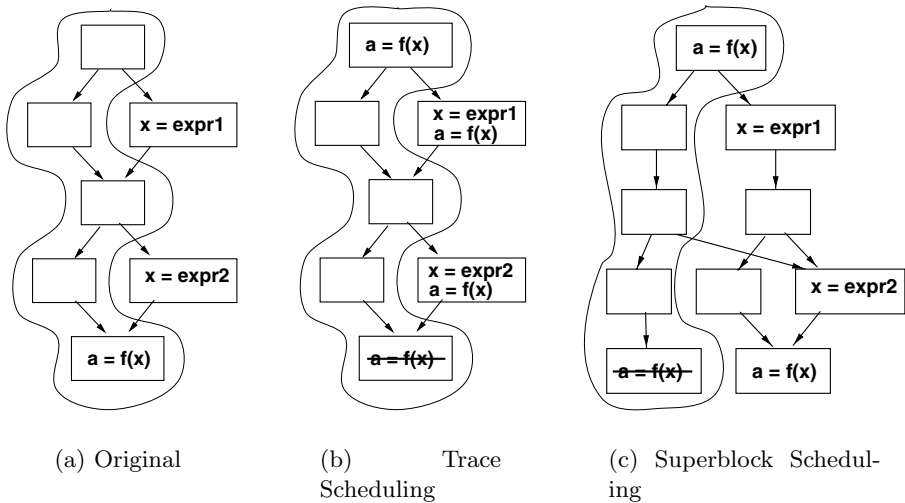


Fig. 3. Comparison of code growth

Figure 3(b) shows an example where an operation is duplicated many times as it moves above a number of control flow joins. The operation moves up the control flow graph along the most likely path, and at each join a copy is added to the incoming off-path edge. Note that in many cases some or all of these duplicates can be eliminated [FGL94]. Many compilers, such as Chameleon and the Multiflow Trace compiler, add a redundancy elimination pass after compensation code generation which removes operations which recompute an existing value. In this example, however, the duplicates cannot be eliminated because the register x is redefined on each of the incoming paths.

The worst case code growth for tail duplication is substantially less. Recall that the trace selection phase results in each basic block in the acyclic region

being placed in exactly one trace. During superblock formation, incoming join edges that allow entry to the trace at any point other than the start are eliminated by duplicating the blocks in the trace which appear after the join. Given that each block is in only one trace, we can be sure that each block will be duplicated at most one time.

Figure 3(c) shows the effect of moving the same operation during Superblock Scheduling. Note that the operation $a = f(x)$ is duplicated only once in the process of moving it from the last basic block to the first, whereas Trace Scheduling (figure 3(b)) requires that it is duplicated twice. Superblock formation never causes an operation to be duplicated more than once. Thus for an acyclic region containing n operations, after superblock formation the region will contain at most $2n$ operations.

It is important to bear in mind that there can be an enormous difference between average and worst case code growth. Global instruction scheduling algorithms similar to Trace Scheduling very rarely result in exponential code growth [Gre00, PSM97]. In most cases we would expect code size increases to be modest, and considerably less than that of tail duplication. Tail duplication, on the other hand, will often cause close to its worst case code growth, as it always duplicates blocks after joins whether or not the instructions following the join will be moved above it or not.

6 Speedup

In general, we would expect the speedup to be similar whether tail duplication or compensation code is used. Both approaches have advantages, however, that may result in better code in some circumstances. For example, tail duplication often makes other optimisations more effective, by allowing them to concentrate on a single superblock, regardless of definitions or usages on other control flow paths. For example Hwu et al [HMC⁺93] found that applying superblock formation made other traditional optimisations significantly more effective even without sophisticated scheduling¹.

Compensation code has an advantage over tail duplication when scheduling off-trace paths. Recall that in figure 3(b) we showed that Trace Scheduling can result in more compensation copies being generated. The two copies of operation $a = f(x)$ are absorbed into other traces and scheduled as part of these traces. The situation with tail duplication and Superblock Scheduling is different. In figure 3(c) we see that the copy of $a = f(x)$ remains in a separate basic block after a join. This operation cannot be scheduled with the operations before the join, since superblocks cannot extend across a join. Thus, if an off-trace path is followed frequently, Superblock Scheduling may produce slower schedules.

¹ It appears that the baseline compiler used for comparison was rather poor. In many cases the reported speedups for Superblock Scheduling were more than four on a 4-issue processor, which is a strong sign that the baseline compiler is producing very poor code.

Code growth also has an important effect on speedup. Instruction cache misses can have a much greater effect on performance than a small difference in the length of the schedule of a rarely followed path. In general we expect that compensation code will produce less code growth, and so is likely to give better instruction cache performance.

7 Experimental Results

We implemented Trace Scheduling and Superblock Scheduling² in the Chameleon Compiler and ILP test-bed. Chameleon provides a highly optimizing ILP compiler for IBM's Tree VLIW architecture. The compiler performs many sophisticated traditional and ILP increasing optimizations. We compiled and scheduled several small benchmark programs whose inner loops contain branches, and four larger programs (*anagram*, *yacr-2*, *ks* and *bc*) with various inner loops. We ran the resulting programs on Chameleon's VLIW machine simulator.

The Chameleon compiler uses an existing C compiler (in our case *gcc*) as a front end to generate the original sequential machine code. The baseline for our speedup calculation is the number of cycles it takes to run this code on a simulated one ALU VLIW machine. This sequential code is then optimised to increase ILP, rescheduled by our global scheduling phase, and registers are re-allocated. We run this scheduled code on a simulated machine which can execute four ALU operations per cycle. The speedup is the baseline cycles, divided by the number needed by the 4-ALU VLIW.

Benchmark	Description
eight	Solve eight queens problem
wc	Unix word count utility
bubble	Bubble sort array of random integers
binsearch	Binary search sorted array
tree	Tree sort array of random integers
branch	Inner loop branch test program
eqn	Inner loop of eqntott
anagram	Generate anagrams of strings
ks	Graph partitioning tool
yacr-2	Channel router
bc	GNU bc calculator

Figure 4 shows the speedup achieved by both algorithms on the benchmarks. As expected, the performance of the two algorithms is very similar, because both concentrate on scheduling the most common path. Where branches are

² For control purposes, we also tested applying Trace Scheduling after superblock formation, and Superblock Scheduling without superblock formation. The results didn't contain any surprises, and are thus not presented here.

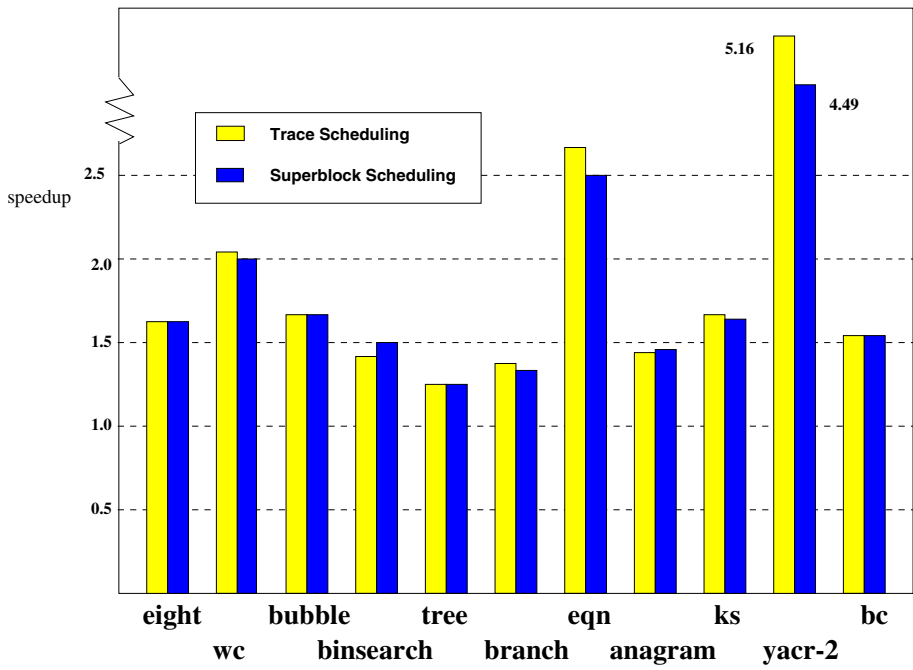


Fig. 4. Speedup

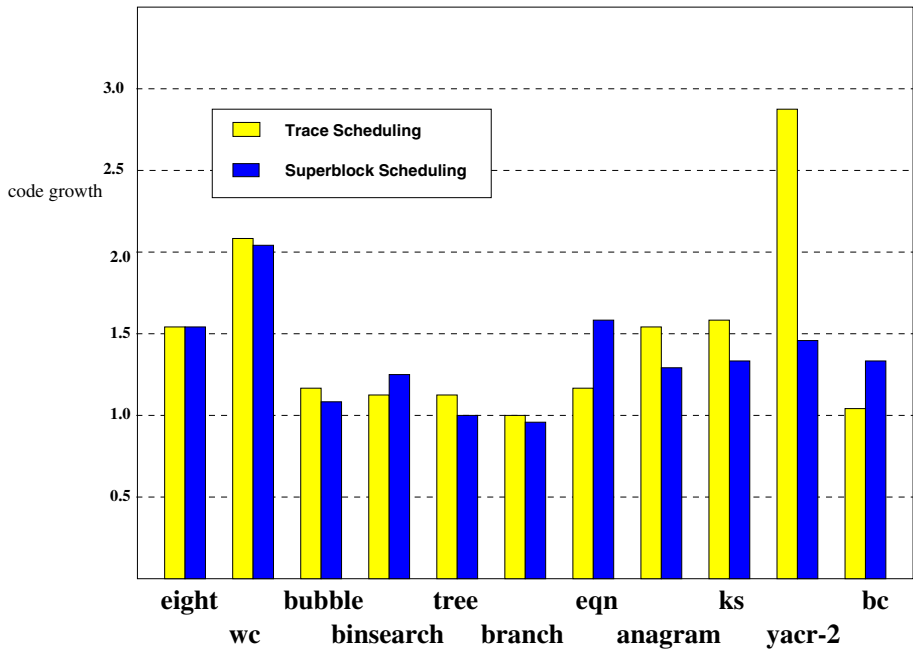


Fig. 5. Code Growth

very unpredictable, Trace Scheduling sometimes does a little better (in *eqn* for example) for reasons outlined in section 6. One remarkable result is for *yacr-2*, where the speedup is greater than the width of the VLIW. In this case, the benefit from other optimisations applied by the VLIW compiler are greater than the gain from global instruction scheduling. In effect, the VLIW optimiser is making up for poor optimisation by the front-end compiler. On investigation, we found that some of the code size increasing optimisations that the VLIW compiler makes are applied to a lesser degree if the original code size is larger. Thus, superblock formation before optimisation can cause these optimisations to be applied less. This is the main reason that Trace Scheduling outperforms Superblock scheduling on *yacr-2*.

The code growth results shown in figure 5 are much more interesting. As expected, Trace Scheduling produces less code growth than Superblock Scheduling for *binsearch* and *eqn* and *bc*. In the case of *eight* the code growth is identical — almost all operations are moved above the joins, so there is no difference between the two approaches. What is surprising is that Trace Scheduling produces more code growth for many benchmarks. In the case of *wc* this arises from its complicated control flow in the inner loop, which causes close to worst case code growth similar to that in figure 3(b). Both *anagram* and *ks* contain similarly complicated control flow, although not in their most common inner loops. Again for *yacr-2* we found that other code size increasing optimisations are primarily responsible for the code growth. There is one particularly large function with complex control flow in *yacr-2* which causes more than average code growth with compensation code. But the great majority of the difference is accounted for by other optimisations and the result shown does not tell us much useful about compensation code or tail duplication.

Looking at the speedup and code growth figures together, it is possible to reach more general conclusions. Trace Scheduling is sometimes a little better and sometimes a little worse than Superblock Scheduling. Contrary to what we expected, compensation code does not produce consistently less code growth than tail duplication. In fact, it often produces more code growth. Given the considerably greater engineering work in implementing compensation code, our results suggest that unless it can be improved in the future, it is not worth the effort.

8 Related Work

Most work in global scheduling concentrates on proposing new schemes, but a number of authors have compared their scheme with an existing one. Moudgill [Mon94] examined a number of different acyclic scheduling regions such as DAG and single path for Fortran programs. In most cases the benefit was small. Havanki et al [WHC98] compared tree scheduling with Superblock Scheduling and reported some speedups for integer code. Mahlke [Mah96] measured the performance improvement of predicated code over Superblock Scheduling. Perhaps the most developed work on comparing strategies is [PSM97] which looks at a

number of issues such as memory disambiguation, probability guided scheduling, single path versus multipath, and software pipelining versus loop unrolling. We are not aware of any existing work which looks at the trade-offs between tail duplication and compensation code.

9 Conclusions

We have examined the problem of maintaining the correctness of code when moving operations above control flow joins during single path global instruction scheduling. Two techniques for doing this are generating compensation code and tail duplication. Compensation code is more complicated and difficult to implement, but offers the possibility of less code growth. We implemented two global scheduling algorithms which use the two techniques in a highly optimising VLIW compiler. Experimental results show that compensation code does not always create less code growth and often creates more. Given the much greater engineering effort in implementing compensation code, it is unlikely to be worth the effort, unless improvements can be made in the future.

Acknowledgments. We would like to thank the VLIW group at IBM's T. J. Watson Research Center for providing us with the Chameleon experimental test-bed. Special thanks to Mayan Moudgill and Michael Gschwind. We are also grateful to Sylvain Lelait for his comments on an earlier version of this work.

References

- CCMmWH91. Pohua P. Chang, William Y. Chen, Scott A. Mahlke, and Wen mei W. Hwu. Impact: An architectural framework for multiple-instruction-issue processors. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 266–275. IEEE, May 1991.
- Ell85. John R. Ellis. *Bulldog: A Compiler for VLIW Architectures*. MIT Press, 1985.
- FERN84. Joseph A. Fisher, John R. Ellis, John C. Ruttenberg, and Alexandru Nicolau. Parallel processing: A smart compiler and a dumb machine. In *ACM '84 Symposium on Compiler Construction*, pages 37–47. ACM, June 1984.
- FGL94. Stefan M. Freudenberger, Thomas R. Gross, and P. Geoffrey Lowney. Avoidance and suppression of compensating code in a trace scheduling compiler. *ACM Transactions on Programming Languages and Systems*, 16(4):1156–1214, 1994.
- Fis81. Joseph A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 30(7):478–490, July 1981.
- Gre00. David Gregg. Global software pipelining with iteration preselection. In *CC-2000 International Conference on Compiler Construction*, LNCS 1781, March 2000.

- HMC⁺93. Wen-mei Hwu, Scott Mahlke, William Chen, Pohua Chang, Nancy Warter, Roger Bringmann, Roland Oullette, Richard Hank, Tokuzo Kiyohara, Grant Haab, John Holm, and Daniel Lavery. The superbloc: An effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing*, pages 229–248, 1993.
- Mah96. Scott Alan Mahlke. *Exploiting Instruction Level Parallelism in the Presence of Conditional Branches*. PhD thesis, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, 1996.
- MCH⁺92. Scott A. Mahlke, William Y. Chen, Wen-mei W. Hwu, B. Ramakrishna Rau, and Michael S. Schlansker. Sentinel scheduling for VLIW and superscalar processors. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 238–247. ACM, September 1992.
- Mou94. Mayan Moudgill. *Implementing and Exploiting Static Speculation and Multiple Issue Processors*. PhD thesis, Cornell University, May 1994.
- Mou97. Mayan Moudgill. Source code of the chameleon compiler. The source code is the only written description of the EPS++ algorithm, 1997.
- PSM97. S. Park, S. Shim, and S. Moon. Evaluation of scheduling techniques on a sparc-based VLIW testbed. In *30th International Symposium on Microarchitecture*. IEEE, November 1997.
- WHC98. S. Banerjia W. Havanki and T. Conte. Treeregion scheduling for wide-issue processors. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture (HPCA-4)*, February 1998.

Register Saturation in Superscalar and VLIW Codes

Sid Ahmed Ali Touati

INRIA, Domaine de Voluceau, BP 105. 78153, Le Chesnay cedex, France
`Sid-Ahmed-Ali.Touati@inria.fr`

Abstract. The registers constraints can be taken into account during the scheduling phase of an acyclic data dependence graph (DAG): any schedule must minimize the register requirement. In this work, we mathematically study and extend the approach which consists of computing the exact upper-bound of the register need for all the valid schedules, independently of the functional unit constraints. A previous work (URSA) was presented in [54]. Its aim was to add some serial arcs to the original DAG such that the worst register need does not exceed the number of available registers. We write an appropriate mathematical formalism for this problem and extend the DAG model to take into account delayed read from and write into registers with multiple registers types. This formulation permits us to provide in this paper better heuristics and strategies (nearly optimal), and we prove that the URSA technique is not sufficient to compute the maximal register requirement, even if its solution is optimal.

1 Introduction and Motivation

In Instruction Level Parallelism (ILP) compilers, code scheduling and register allocation are two major tasks for code optimization. Code scheduling consists of maximizing the exploitation of the ILP offered by the code. One factor that inhibits such use is the registers constraints. A limited number of registers prohibits an unbounded number of values simultaneously alive. If the register allocation is carried out before scheduling, false dependencies are introduced because of the registers reuse, producing a negative impact on the available ILP exposed to the scheduler. If scheduling is carried out before, spill code might be introduced because of an insufficient number of registers. A better approach is to make code scheduling and register allocation interact with each other in a complex combined pass, making the register allocation and the scheduling heuristics very correlated.

In this article, we present our contribution to avoiding an excessive number of values simultaneously alive for all the valid schedules of a DAG, previously studied in [45]. Our pre-pass analyzes a DAG (with respect to control flow) to deduce the maximal register need for all schedules. We call this limit *the register saturation* (RS) because the register need can reach this limit but never exceed it. We provide better heuristics to compute and reduce it if it exceeds the number

of available registers by introducing new arcs. Experimental results show that in most cases our strategies are nearly optimal.

This article is organized as follows. Section 2 presents our DAG model which can be used for both superscalar and VLIW processors (UAL and NUAL semantics [15]). The RS problem is theoretically studied in Sect. 3. If it exceeds the number of available registers, a heuristic for reducing it is given in Sect. 4. We have implemented software tools, and experimental results are described in Sect. 5. Some related work in this field is given in Sect. 6. We conclude with our remarks and perspectives in Sect. 7.

2 DAG Model

A DAG $G = (V, E, \delta)$ in our study represents the data dependences between the operations and any other serial constraints. Each operation u has a strictly positive latency $lat(u)$. The DAG is defined by its set of operations V , its set of arcs $E = \{(u, v) \mid u, v \in V\}$, and δ such that $\delta(e)$ is the latency of the arc e in terms of processor clock cycles.

A schedule σ of G is a positive function which gives an integer execution (issue) time for each operation :

$$\sigma \text{ is valid} \iff \forall e = (u, v) \in E \quad \sigma(v) - \sigma(u) \geq \delta(e)$$

We note by $\Sigma(G)$ the set of *all* the valid schedules of G . Since writing to and reading from registers could be delayed from the beginning of the operation schedule time (VLIW case), we define the two delay functions δ_r and δ_w such that $\delta_w(u)$ is the write cycle of the operation u , and $\delta_r(u)$ is the read cycle of u . In other words, u reads from the register file at instant $\sigma(u) + \delta_r(u)$, and writes in it at instant $\sigma(u) + \delta_w(u)$.

To simplify the writing of some mathematical formulas, we assume that the DAG has one source (\top) and one sink (\perp). If not, we introduce two fictitious nodes (\top, \perp) representing nops (evicted at the end of the RS analysis). We add a virtual serial arc $e_1 = (\top, s)$ to each source with $\delta(e_1) = 0$, and an arc $e_2 = (t, \perp)$ from each sink with the latency of the sink operation $\delta(e_2) = lat(t)$. The total schedule time of a schedule is then $\sigma(\perp)$. The null latency of an added arc e_1 is not inconsistent with our assumption that latencies must be strictly positive because the added virtual serial arcs no longer represent data dependencies. Furthermore, we can avoid introducing these virtual nodes without any consequence on our theoretical study since their purpose is only to simplify some mathematical expressions.

When studying the register need in a DAG, we make a difference between the nodes, depending on whether they define a value to be stored in a register or not, and also depending on which register type we are focusing on (int, float, etc.). We also make a difference between edges, depending on whether they are flow dependencies through the registers of the type considered :

- $V_R \subseteq V$ is the subset of operations which define a value of the type under consideration (int, float, etc.), we simply call them *values*. We assume that

at most one value of the type considered can be defined by an operation. The operations which define multiple values are taken into account if they define at most one value of the type considered.

- $E_R \subseteq E$ is the subset of arcs representing true dependencies through a value of the type considered. We call them *flow arcs*.
- $E_S = E - E_R$ are called *serial arcs*.

Figure 1b gives the DAG that we use in this paper constructed from the code of part (a). In this example, we focus on the floating point registers: the values and flow arcs are shown by bold lines. We assume for instance that each read occurs exactly at the schedule time and each write at the final execution step ($\delta_r(u) = 0$, $\delta_w(u) = \text{lat}(u) - 1$).

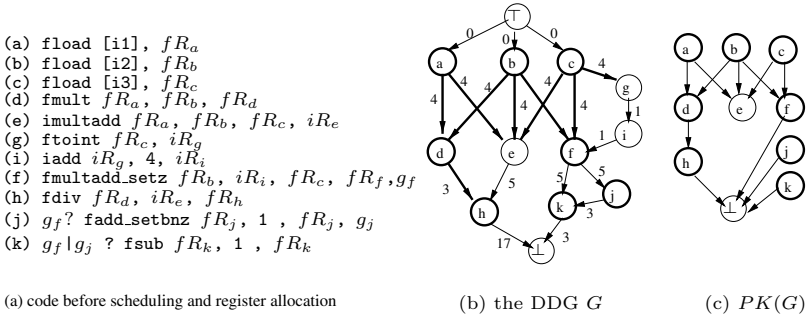


Fig. 1. DAG model

Notation and Definitions on DAGs

In this paper, we use the following notations for a given DAG $G = (V, E)$:

- $\Gamma_G^+(u) = \{v \in V / (u, v) \in E\}$ successors of u ;
- $\Gamma_G^-(u) = \{v \in V / (v, u) \in E\}$ predecessors of u ;
- $\forall e = (u, v) \in E$ $\text{source}(e) = u \wedge \text{target}(e) = v$. u, v are called *endpoints*;
- $\forall u, v \in V$: $u < v \iff \exists$ a path (u, \dots, v) in G ;
- $\forall u, v \in V$: $u \parallel v \iff \neg(u < v) \wedge \neg(v < u)$. u and v are said to be *parallel*;
- $\forall u \in V$ $\uparrow u = \{v \in V / v = u \vee v < u\}$ u 's ascendants including u ;
- $\forall u \in V$ $\downarrow u = \{v \in V / v = u \vee u < v\}$ u 's descendants including u .
- two arcs e, e' are *adjacent* iff they share an endpoint;
- $A \subseteq V$ is an *antichain* in $G \iff \forall u, v \in A$ $u \parallel v$;
- AM is a *maximal antichain* $\iff \forall A$ antichain in G $|A| \leq |AM|$;
- the *extended DAG* $G \setminus^{E'}$ of G generated by the arcs set $E' \subseteq V^2$ is the graph obtained from G after adding the arcs in E' . As a consequence, any valid schedule of G' is necessarily a valid schedule for G :

$$G' = G \setminus^{E'} \implies \Sigma(G') \subseteq \Sigma(G)$$

- let $I_1 = [a_1, b_1] \subset \mathbb{N}$ and $I_2 = [a_2, b_2] \subset \mathbb{N}$ be two integer intervals. We say that I_1 is before I_2 , noted by $I_1 \prec I_2$, iff $b_1 < a_2$.

3 Register Saturation

3.1 Register Need of a Schedule

Given a DAG $G = (V, E, \delta)$, a value $u \in V_R$ is alive just after the writing clock cycle of u until its last reading (consumption). The values which are not read in G or are still alive when exiting the DAG must be kept in registers. We handle these values by considering that the bottom node \perp consumes them. We define the set of consumers for each value $u \in V_R$ as

$$Cons(u) = \begin{cases} \{v \in V / (u, v) \in E_R\} & \text{if } \exists (u, v) \in E_R \\ \perp & \text{otherwise} \end{cases}$$

Given a schedule $\sigma \in \Sigma(G)$, the last consumption of a value is called the killing date and noted :

$$\forall u \in V_R \quad kill_\sigma(u) = \max_{v \in Cons(u)} (\sigma(v) + \delta_r(v))$$

All the consumers of u whose reading time is equal to the killing date of u are called the killers of u . We assume that a value written at instant t in a register is available one step later. That is to say, if operation u reads from a register at instant t while operation v is writing in it at the same time, u does not get v 's result but gets the value previously stored in this register. Then, the *life interval* L_u^σ of a value u according to σ is $[\sigma(u) + \delta_w(u), kill_\sigma(u)]$.

Given the life intervals of all the values, the register need of σ is the maximum number of values simultaneously alive :

$$RN_\sigma(G) = \max_{0 \leq i \leq \sigma(\perp)} |vsa_\sigma(i)|$$

where $vsa_\sigma(i) = \{u \in V_R / i \in L_u^\sigma\}$ is the set of values alive at time i

Building a register allocation (assign a physical register to each value) with R available registers for a schedule which needs R registers can be easily done with a polynomial-complexity algorithm without introducing spill code nor increasing the total schedule time [16].

3.2 Register Saturation Problem

The RS is the maximal register need for all the valid schedules of the DAG :

$$RS(G) = \max_{\sigma \in \Sigma(G)} RN_\sigma(G)$$

We call σ a *saturating schedule* iff $RN_\sigma(G) = RS(G)$. In this section, we study how to compute $RS(G)$. We will see that this problem comes down to

answering the question “*which operation must kill this value ?*” When looking for saturating schedules, we do not worry about the total schedule time. Our aim is only to prove that the register need can reach the RS but cannot exceed it. Minimizing the total schedule time is considered in Sect. 4 when we reduce the RS. Furthermore, for the purpose of building saturating schedules, we have proven in [16] that to maximize the register need, looking for only one suitable killer of a value is sufficient rather than looking for a group of killers: for any schedule that assigns more than one killer for a value, we can obviously build another schedule with at least the same register need such that this value is killed by only one consumer. So, the purpose of this section is to select a suitable killer for each value to saturate the register requirement.

Since we do not assume any schedule, the life intervals are not defined so we cannot know at which date a value is killed. However, we can deduce which consumers in $Cons(u)$ are impossible killers for the value u . If $v_1, v_2 \in Cons(u)$ and \exists a path $(v_1 \cdots v_2)$, v_1 is always scheduled before v_2 with at least $lat(v_1)$ processor cycles. Then v_1 can never be the last read of u (remember that we assume strictly positive latencies). We can consequently deduce which consumers can “potentially” kill a value (possible killers). We note $pkill_G(u)$ the set of the operations which can kill a value $u \in V_R$:

$$pkill_G(u) = \{v \in Cons(u) / \downarrow v \cap Cons(u) = \{v\}\}$$

One can check that all operations in $pkill_G(u)$ are parallel in G . Any operation which does not belong to $pkill_G(u)$ can never kill the value u .

Lemma 1. *Given a DAG $G = (V, E, \delta)$, then $\forall u \in V_R$*

$$\forall \sigma \in \Sigma(G) \quad \exists v \in pkill_G(u) : \quad \sigma(v) + \delta_r(v) = kill_\sigma(u) \quad (1)$$

$$\forall v \in pkill_G(u) \quad \exists \sigma \in \Sigma(G) : \quad kill_\sigma(u) = \sigma(v) + \delta_r(v) \quad (2)$$

Proof. A complete proof is given in [17], page 13.

A *potential killing DAG* of G , noted $PK(G) = (V, E_{PK})$, is built to model the potential killing relations between operations, (see Fig. 1c), where:

$$E_{PK} = \{(u, v) / u \in V_R \wedge v \in pkill_G(u)\}$$

There may be more than one operation candidate for killing a value. Let us begin by assuming a *killing function* which enforces an operation $v \in pkill_G(u)$ to be the killer of $u \in V_R$. If we assume that $k(u)$ is the unique killer of $u \in V_R$, we must always verify the following assertion:

$$\forall v \in pkill_G(u) - \{k(u)\} \quad \sigma(v) + \delta_r(v) < \sigma(k(u)) + \delta_r(k(u)) \quad (3)$$

There is a family of schedules which ensures this assertion. To define them, we extend G by new serial arcs that enforce all the potential killing operations of each value u to be scheduled before $k(u)$. This leads us to define an extended DAG associated to k noted $G_{\rightarrow k} = G \setminus^{E_k}$ where:

$$E_k = \left\{ e = (v, k(u)) / u \in V_R \quad v \in pkill_G(u) - \{k(u)\} \quad \text{with } \delta(e) = \delta_r(v) - \delta_r(k(u)) + 1 \right\}$$

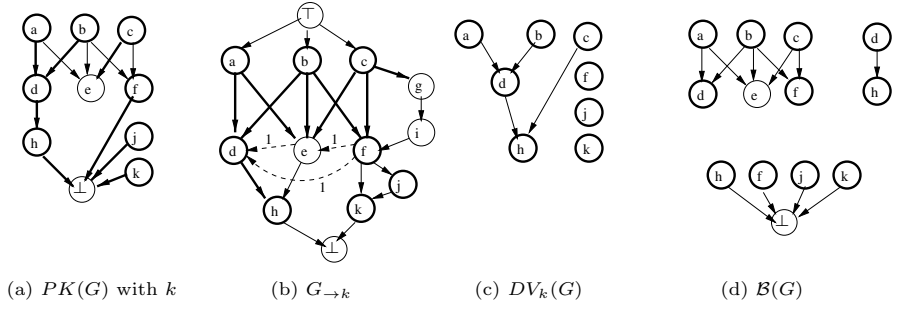


Fig. 2. Valid killing function and bipartite decomposition

Then, any schedule $\sigma \in \Sigma(G_{\rightarrow k})$ ensures Property 3. The condition of the existence of such a schedule defines the condition of a *valid killing function*:

k is a valid killing function $\iff G_{\rightarrow k}$ is acyclic

Figure 2 gives an example of a valid killing function k . This function is shown by bold arcs in part (a), where each target kills its sources. Part (b) is the DAG associated to k .

Provided a valid killing function k , we can deduce the values which can never be simultaneously alive for any $\sigma \in \Sigma(G_{\rightarrow k})$. Let $\downarrow_R(u) = \downarrow u \cap V_R$ be the set of the descendant values of $u \in V$.

Lemma 2. *Given a DAG $G = (V, E, \delta)$ and a valid killing function, then :*

1. *the descendant values of $k(u)$ cannot be simultaneously alive with u :*

$$\forall u \in V_R \quad \forall \sigma \in \Sigma(G_{\rightarrow k}) \quad \forall v \in \downarrow_R k(u) \quad L_\sigma^u \prec L_\sigma^v \quad (4)$$

2. *there exists a valid schedule which makes the other values non descendant of $k(u)$ simultaneously alive with u , i.e. $\forall u \in V_R \quad \exists \sigma \in \Sigma(G_{\rightarrow k})$:*

$$\forall v \in \left(\bigcup_{v' \in pkill_G(u)} \downarrow_R v' \right) - \downarrow_R k(u) \quad L_\sigma^u \cap L_\sigma^v \neq \emptyset \quad (5)$$

Proof. A complete proof is given in [17], page 23.

We define a DAG which models the values that can never be simultaneously alive according to k . The *disjoint value DAG* of G associated to k , and noted $DV_k(G) = (V_R, E_{DV})$ is defined by :

$$E_{DV} = \{(u, v) / u, v \in V_R \wedge v \in \downarrow_R k(u)\}$$

Any arc (u, v) in $DV_k(G)$ means that u 's life interval is always before v 's life interval according to any schedule of $G_{\rightarrow k}$, see part Fig. 2.d¹. This definition permits us to state in the following theorem that the register need of any schedule of $G_{\rightarrow k}$ is always less than or equal to a maximal antichain in $DV_k(G)$. Also, there is always a schedule which makes all the values in this maximal antichain simultaneously alive.

Theorem 1. *Given a DAG $G = (V, E, \delta)$ and a valid killing function k then :*

- $\forall \sigma \in \Sigma(G_{\rightarrow k}) : RN_\sigma(G) \leq |AM_k|$
- $\exists \sigma \in \Sigma(G_{\rightarrow k}) : RN_\sigma(G) = |AM_k|$

where AM_k is a maximal antichain in $DV_k(G)$

Proof. A complete proof is given in [17], page 18.

Theorem 1 allows us to rewrite the RS formula as

$$RS(G) = \max_{k \text{ a valid killing function}} |AM_k|$$

where AM_k is a maximal antichain in $DV_k(G)$. We refer to the problem of finding such a killing function as the *maximizing maximal antichain* problem (MMA). We call each solution for the MMA problem a *saturating killing function*, and AM_k its *saturating values*. Unfortunately, we have proven in [17] (page 24) that finding a saturating killing function is NP-complete.

3.3 A Heuristic for Computing the RS

This section presents our heuristics to approximate an optimal k by another valid killing function k^* . We have to choose a killing operation for each value such that we maximize the parallel values in $DV_k(G)$. Our heuristics focus on the potential killing DAG $PK(G)$, starting from source nodes to sinks. Our aim is to select a group of killing operations for a group of parents to keep as many descendant values alive as possible. The main steps of our heuristics are :

1. decompose the potential killing DAG $PK(G)$ into connected bipartite components ;
2. for each bipartite component, search for the best saturating killing set (defined below) ;
3. choose a killing operation within the saturating killing set (defined below).

We decompose the potential killing DAG into connected bipartite components (CBC) in order to choose a common saturating killing set for a group of parents. Our purpose is to have a maximum number of children and their descendants values simultaneously alive with their parents values. A CBC $cb = (S_{cb}, T_{cb}, E_{cb})$ is a partition of a subset of operations into two disjoint sets where :

¹ This DAG is simplified by transitive reduction.

- $E_{cb} \subseteq E_{PK}$ is a subset of the potential killing relations ;
- $S_{cb} \subseteq V_R$ is the set of the parent values, such that each parent is killed by at least one operation in T_{cb} ;
- $T_{cb} \subset V$ is the set of the children, such that any operation in T_{cb} can potentially kill at least a value in S_{cb} .

A bipartite decomposition of the potential killing graph $PK(G)$ is the set (see Fig. 2.d)

$$\mathcal{B}(G) = \{cb = (S_{cb}, T_{cb}, E_{cb}) / \forall e \in E_{PK} \exists cb \in \mathcal{B}(G) : e \in E_{cb}\}$$

Note that $\forall cb \in \mathcal{B}(G) \forall s, s' \in S_{cb} \forall t, t' \in T_{cb} \quad s || s' \wedge t || t'$ in $PK(G)$.

A saturating killing set $SKS(cb)$ of a bipartite component $cb = (S_{cb}, T_{cb}, E_{cb})$ is a subset $T'_{cb} \subseteq T_{cb}$ such that if we choose a killing operation from this subset, then we get a maximal number of descendant values of children in T_{cb} simultaneously alive with parent values in S_{cb} .

Definition 1 (Saturating Killing Set). *Given a DAG $G = (V, E, \delta)$, a saturating killing set $SKS(cb)$ of a connected bipartite component $cb \in \mathcal{B}(G)$ is a subset $T'_{cb} \subseteq T_{cb}$, such that :*

1. *killing constraints :*

$$\bigcup_{t \in T'_{cb}} \Gamma_{cb}^-(t) = S_{cb}$$

2. *minimizing the number of descendant values of T'_{cb}*

$$\min | \bigcup_{t \in T'_{cb}} \downarrow_R t |$$

Unfortunately, computing a SKS is also NP-complete ([17], page 81).

A Heuristic for Finding a SKS Intuitively, we should choose a subset of children in a bipartite component that would kill the greatest number of parents while minimizing the number of descendant values. We define a cost function ρ that enables us to choose the best candidate child. Given a bipartite component $cb = (S_{cb}, T_{cb}, E_{cb})$ and a set Y of (cumulated) descendant values and a set X of non (yet) killed parents, the cost of a child $t \in T_{cb}$ is :

$$\rho_{X,Y}(t) = \begin{cases} \frac{|\Gamma_{cb}^-(t) \cap X|}{|\downarrow_R t \cup Y|} & \text{if } \downarrow_R t \cup Y \neq \phi \\ |\Gamma_{cb}^-(t) \cap X| & \text{otherwise} \end{cases}$$

The first case enables us to select the child which covers the most non killed parents with the minimum descendant values. If there is no descendant value, then we choose the child that covers the most non killed parents.

Algorithm 1 gives a modified greedy heuristic that searches for an approximation SKS^* and computes a killing function k^* in polynomial time. Our heuristic has the following properties.

Algorithm 1 Greedy- k : a heuristics for the MMA problem

Require: a DAG $G = (V, E, \delta)$

for all values $u \in V_R$ **do**

$k^*(u) = \perp$ {all values are initially non killed}

end for

build $\mathcal{B}(G)$ the bipartite decomposition of $PK(G)$.

for all bipartite component $cb = (S_{cb}, T_{cb}, E_{cb}) \in \mathcal{B}(G)$ **do**

$X := S_{cb}$ {all parents are initially uncovered}

$Y := \phi$ {initially, no cumulated descendant values}

$SKS^*(cb) := \phi$

while $X \neq \phi$ **do** {build the SKS for cb }

select the child $t \in T_{cb}$ with the maximal cost $\rho_{X,Y}(t)$

$SKS^*(cb) := SKS^*(cb) \cup \{t\}$

$X := X - \Gamma_{cb}^-(t)$ {remove covered parents}

$Y := Y \cup \downarrow_R t$ {update the cumulated descendent values}

end while

for all $t \in SKS^*(cb)$ **do** {in decreasing cost order}

for all parent $s \in \Gamma_{cb}^-(t)$ **do**

if $k^*(s) = \perp$ **then** {kill non killed parents of t }

$k^*(s) := t$

end if

end for

end for

end for

Theorem 2. *Given a DAG $G = (V, E, \delta)$, then :*

1. *Greedy- k always produces a valid killing function k^* ;*
2. *$PK(G)$ is an inverted tree \implies Greedy- k is optimal.*

Proof. Complete proofs for both (1) and (2) are given in [17], pages 31 and 44 resp.

Since the approximated killing function k^* is valid, Theorem 1 ensures that we can always find a valid schedule which requires exactly $|AM_{k^*}|$ registers. As consequence, our heuristic does not compute an upper bound of the optimal register saturation and then the optimal RS can be greater than the one computed by Greedy- k . A conservative heuristic which computes a solution exceeding the optimal RS cannot ensure the existence of a valid schedule which reaches the computed limit, and hence it would imply an obsolete RS reduction process and a waste of registers. The validity of a killing function is a key condition because it ensures that there exists a register allocation with exactly $|AM_{k^*}|$ registers. As summary, here are our steps to compute the RS :

1. apply Greedy- k on G . The result is a valid killing function k^* ;
2. construct the disjoint value DAG $DV_{k^*}(G)$;
3. find a maximal antichain AM_{k^*} of $DV_{k^*}(G)$ using Dilworth decomposition [10] ; Saturating values are then AM_{k^*} and $RS^*(G) = |AM_{k^*}| \leq RS(G)$.

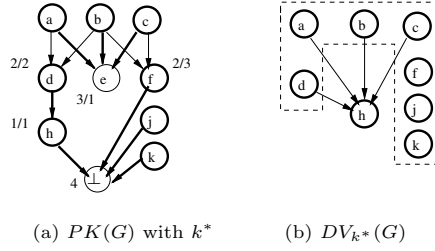


Fig. 3. Example of computing the register saturation

Figure 3a shows a saturating killing function k^* computed by Greedy- k : bold arcs denote that each target kills its sources. Each killer is labeled by its cost ρ . Part (b) gives the disjoint value DAG associated to k^* . The Saturating values are $\{a, b, c, d, f, j, k\}$, so the RS is 7.

4 Reducing the Register Saturation

In this section we build an extended DAG $\overline{G} = G \setminus \overline{E}$ such that the RS is limited by a strictly positive integer (number of available registers) with the respect of the critical path. Let \mathcal{R} be this limit. Then :

$$\forall \sigma \in \Sigma(\overline{G}) : RN_{\sigma}(\overline{G}) \leq RS(\overline{G}) \leq \mathcal{R}$$

We have proven in [16] that this problem is NP-hard. In this section we present a heuristic that adds serial arcs to prevent some saturating values in AM_k (according to a saturating killing function k) from being simultaneously alive for any schedule. Also, we must care to not increase the critical path if possible.

Serializing two values $u, v \in V_R$ means that the kill of u must always be carried out before the definition of v , or vice-versa. A value serialization $u \rightarrow v$ for two values $u, v \in V_R$ is defined by :

- if $v \in pkill_G(u)$ then add the serial arcs $\{e = (v', v) /$

$$v' \in pkill_G(u) - \{v\} \text{ with } \delta(e) = \delta_r(v') - \delta_w(v)\}$$

- else add the serial arcs $\{e = (u', v) /$

$$u' \in pkill_G(u) \wedge \neg(v < u') \text{ with } \delta(e) = \delta_r(u') - \delta_w(v)\}$$

To do not violate the DAG property (we must not introduce a cycle), some serializations must be filtered out. The condition for applying $u \rightarrow v$ is that $\forall v' \in pkill_G(u) : \neg(v < v')$. We chose the best serialization within the set of all the possible ones by using a cost function $\omega(u \rightarrow v) = (\omega_1, \omega_2)$, such that :

- $\omega_1 = \mu_1 - \mu_2$ is the prediction of the reduction obtained within the saturating values if we carry out this value serialization, where

- μ_1 is the number of saturating values serialized after u if we carry out the serialization;
- μ_2 is the predicted number of u 's descendant values that can become simultaneously alive with u ;
- ω_2 is the increase in the critical path.

Our heuristic is described in Algorithm 2. It iterates the value serializations within the saturating values until we get the limit \mathcal{R} or until no more serializations are possible (or none is expected to reduce the RS). One can check that if there is no possible value serialization in the original DAG, our algorithm exits at the first iteration of the outer while-loop. If it succeeds, then any schedule of \overline{G} needs at most \mathcal{R} registers. If not, it still decreases the original RS, and thus limits the register need. Introducing and minimizing the spill code is another NP-complete problem studied in [8, 3, 2, 9, 14] and not addressed in this work.

Now, we explain how to compute the prediction parameters μ_1, μ_2, ω_2 . We note \overline{G}_i the extended DAG of step i , k_i its saturating function, and AM_{k_i} its saturating values and $\downarrow_{R_i} u$ the descendant values of u in \overline{G}_i :

1. $(u \rightarrow v)$ ensures that $k_{i+1}(u) < v$ in \overline{G}_{i+1} . According to Lemma 2, $\mu_1 = |\downarrow_{R_i} v \cap AM_{k_i}|$ is the number of saturating values in \overline{G}_i which cannot be simultaneously alive with u in \overline{G}_{i+1} ;
2. new saturating values could be introduced into \overline{G}_{i+1} : if $v \in \text{pkill}_{\overline{G}_i}(u)$, we force $k_{i+1}(u) = v$. According to Lemma 2,

$$\mu_2 = \left| \left(\bigcup_{v' \in \text{pkill}_{\overline{G}_i}(u)} \downarrow_{R_i} v' \right) - \downarrow_{R_i} v \right|$$

is the number of values which could be simultaneously alive with u in \overline{G}_{i+1} . $\mu_2 = 0$ otherwise;

3. if we carry out $(u \rightarrow v)$ in \overline{G}_i , the introduced serial arcs could enlarge the critical path. Let $lp_i(v', v)$ be the longest path going from v' to v in \overline{G}_i . The new longest path in \overline{G}_{i+1} going through the serialized nodes is:

$$\max_{\substack{\text{introduced } e=(v',v) \\ \delta(e) > lp_i(v',v)}} lp_i(\top, v') + lp_i(v, \perp) + \delta(e)$$

If this path is greater than the critical path in \overline{G}_i , then ω_2 is the difference between them, 0 otherwise.

At the end of the algorithm, we apply a general verification step to ensure the potential killing property proven in Lemma 4 for the original DAG. We have proven in Lemma 4 that the operations which do not belong to $\text{pkill}_G(u)$ cannot kill the value u . After adding the serial arcs to build \overline{G} , we might violate this assertion because we introduce some arcs with negative latencies. To overcome this problem, we must guarantee the following assertion: $\forall u \in V_R, \forall v' \in \text{Cons}(u) - \text{pkill}_{\overline{G}}(u)$

$$\exists v \in \text{pkill}_{\overline{G}}(u)/v' < v \text{ in } \overline{G} \implies lp_{\overline{G}}(v', v) > \delta_r(v') - \delta_r(v) \quad (6)$$

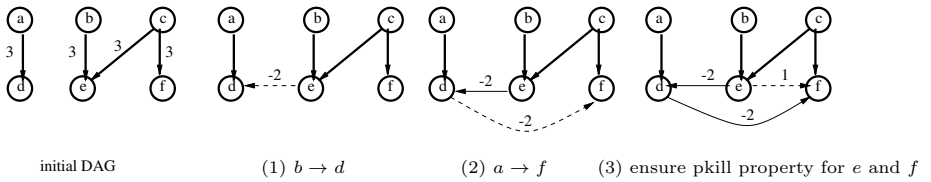
Algorithm 2 Value Serialization Heuristic**Require:** a DAG $G = (V, E, \delta)$ and a strictly positive integer \mathcal{R} $\overline{G} := G$ compute AM_k , saturating values of \overline{G} ;**while** $|AM_k| > \mathcal{R}$ **do** construct the set U_k of all admissible serializations between saturating values in AM_k with their costs (ω_1, ω_2) ; **if** $\nexists (u \rightarrow v) \in U/\omega_1(u \rightarrow v) > 0$ **then** {no more possible RS reduction}

exit;

end if $X := \{(u \rightarrow v) \in U/\omega_2(u \rightarrow v) = 0\}$ {the set of value serializations that do not increase the critical path} **if** $X \neq \emptyset$ **then** choose a value serialization $(u \rightarrow v)$ in X with the minimum cost $\mathcal{R} - \omega_1$; **else** choose a value serialization $(u \rightarrow v)$ in X with the minimum cost ω_2 ; **end if** carry out the serialization $(u \rightarrow v)$ in \overline{G} ; compute the new saturating values AM_k of \overline{G} ;**end while**

ensure potential killing operations property {check longest paths between kill operations}

In fact, this problem occurs if we create a path in \overline{G} from v' to v where $v, v' \in \text{pkill}_G(u)$. If assertion (6) is not verified, we add a serial arc $e = (v', v)$ with $\delta(e) = \delta_r(v') - \delta_r(v) + 1$ as illustrated in Fig. 4.

**Fig. 4.** Check Potential Killers Property

Example 1. Figure 5 gives an example for reducing the RS of our DAG from 7 to 4 registers. We remind that the saturating values of G are $AM_k = \{a, b, c, d, f, j, k\}$. Part (a) shows all the possible value serializations within these saturating values. Our heuristic selects $a \rightarrow f$ as a candidate, since it is expected to eliminate 3 saturating values without increasing the critical path. The maximal introduced longest path through this serialization is $(\top, a, d, f, k, \perp) = 8$, which is less than the original critical path (26). The extended DAG \overline{G} is presented in part (b) where the value serialization $a \rightarrow f$ is introduced: we add the serial arcs (e, f) and (d, f) with a -4 latency. Finally, we add the serial arcs (e, f) and

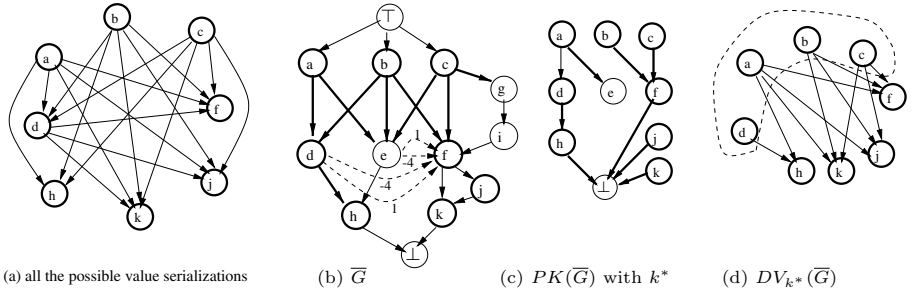


Fig. 5. Reducing register saturation

(d, f) with a unit latency to ensure the $pkill_{\overline{G}}(b)$ property. The whole critical path does not increase and RS is reduced to 4. Part (c) gives a saturating killing function for \overline{G} , shown with bold arcs in $PK(\overline{G})$. $DV_{k^*}(\overline{G})$ is presented in part (d) to show that the new RS becomes 4 floating point registers.

5 Experimentation

We have implemented the RS analysis using the LEDA framework. We carried out our experiments on various floating point numerical loops taken from various benchmarks (livermore, whetstone, spec-fp, etc.). We focus in these codes on the floating point registers. The first experimentation is devoted to checking Greedy- k efficiency. For this purpose, we have defined and implemented in [16] an integer linear programming model to compute the optimal RS of a DAG. We use CPLEX to resolve these linear programming models. The total number of experimented DAGs is 180, where the number of nodes goes up to 120 and the number of values goes up to 114. Experimental results show that our heuristics give quasi-optimal solutions. The worst experimental error is 1, which means that the optimal RS is in worst case greater by one register than the one computed by Greedy- k .

The second experimentation is devoted to checking the efficiency of the value serialization heuristics in order to reduce the RS. We have also defined and implemented in [16] an integer linear programming model to compute the optimal reduction of the RS with a minimum critical path increase (NP-hard problem). The total number of experimented DAGs is 144, where the number of nodes goes up to 80 and the number of values goes up to 76. In almost all cases, our heuristics manages to get the optimal solutions. Optimal reduced RS was in the worst cases less by one register than our heuristics results. Since RS computation in the value serialization heuristics is done by Greedy- k , we add its worst experimental error (1 register) which leads to a total maximal error of two registers. All optimal vs. approximated results are fully detailed in [16].

Since our strategies result in a good efficiency, we use them to study the RS behavior in DAGs. Experimentation on only loop bodies shows that the RS is low, ranging from 1 to 8. We have unrolled these loops with different

unrolling factors going up to 20 times. The aim of such unrolling is to get large DAGs, increase the registers pressure and expose more ILP to hide memory latencies. We carried out a wide range of experiments to study the RS and its reduction with various limits of available registers (going from 1 up to 64). We experimented 720 DAGs where the number of nodes goes up to 400 and the number of values goes up to 380. Full results are detailed in [17,16].

The first remark deduced from our full experiments is that the RS is lower than the number of available registers in a lot of cases. The RS analysis makes it possible to avoid the registers constraints in code scheduling: most of these codes can be scheduled without any interaction with the register allocation, which decreases the compile-time complexity. Second, in most cases our heuristics succeeds in reducing it until reaching the targeted limit. In a few cases we lose some ILP because of the intrinsic register pressure of the DAGs: but since spill code decreases the performance dramatically because of the memory access latencies, a tradeoff between spilling and increasing the overall schedule time can be done in few critical cases. Finally, in the cases where the RS is lower than the number of available registers, we can use the extra non used registers by assigning to them some global variables and array elements with the guarantee that no spill code could be introduced after by the scheduler and the register allocator.

6 Related Work and Discussion

Combining code scheduling and register allocation in DAGs was studied in many works. All the techniques described in [11,6,13,7,12] used their heuristics to build an optimized schedule without exceeding a certain limit of values simultaneously alive. The dual notion of the RS, called the register sufficiency, was studied in [4]. Given a DAG, the authors gave a heuristic which found the minimum register need; the computation was $O(\log^2|V|)$ factor of the optimal. Note that we can easily use the RS reduction to compute the register sufficiency. This is done in practice by setting $\mathcal{R} = 1$ as the targeted limit for the RS reduction.

Our work is an extension to URSA [4,5]. The minimum killing set technique tried to saturate the register requirement in a DAG by keeping the values alive as late as possible: the authors proceeded by keeping as many children alive as possible in a bipartite component by computing the minimum set which killed all the parent's values. First, since the authors did not formalize the RS problem, we can easily give examples to show that a minimum killing set does not saturate the register need, even if the solution is optimal [17]. Figure. 6 shows an example where the RS computed by our heuristics (Part b) is 6 where the optimal solution for URSA yields a RS of 5 (part c). This is because URSA did not take into account the descendant values while computing the killing sets. Second, the validity of the killing functions is an important condition to compute the RS and unfortunately was not included in URSA. We have proven in [17] that non valid killing functions can exist if no care is taken. Finally, the URSA DAG

model did not differentiate between the types of the values and did not take into account delays in reads from and writes into the registers file.

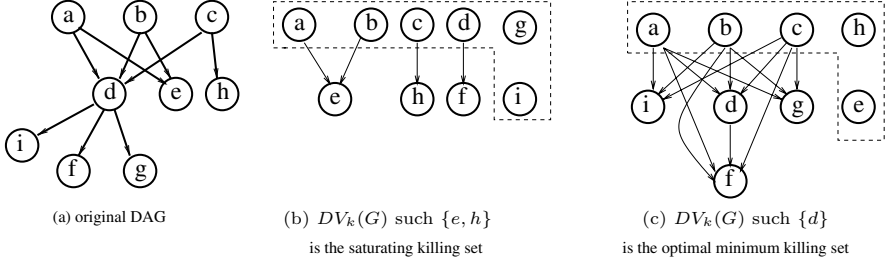


Fig. 6. URSA drawback

7 Conclusion and Future Work

In our work, we mathematically study and define the RS notion to manage the registers pressure and avoid spill code before the scheduling and register allocation passes. We extend URSA by taking into account the operations in both Unit and Non Unit Assumed Latencies (UAL and NUAL [15]) semantics with different types (values and non values) and values (float, integer, etc.). The formal mathematical modeling and theoretical study permit us to give nearly optimal strategies and prove that the minimum killing set is insufficient to compute the RS. Experimentations show that the registers constraints can be obsolete in many codes, and may therefore be ignored in order to simplify the scheduling process. The heuristics we use manage to reduce the RS in most cases while some ILP is lost in few DAGs. We think that reducing the RS is better than minimizing the register need: this is because minimizing the register need increases the register reuse, and the ILP loss must increase as consequence. Our DAG model is sufficiently general to meet all current architecture properties (RISC or CISC), except for some architectures which support issuing dependent instructions at the same clock cycle, which would require representation using null latency. Strictly positive latencies are assumed to prove the pkill operation property (Lemma 1) which is important to build our heuristics. We think that this restriction should not be a major drawback nor an important factor in performance degradation, since null latency operations do not generally contribute to the critical execution paths. In the future, we will extend our work to loops. We will study how to compute and reduce the RS in the case of cyclic schedules like software pipelining (SWP) where the life intervals become circular.

References

1. A. Agrawal, P. Klein, and R. Ravi. Ordering Problems Approximated : Register Sufficiency, Single Processor Scheduling and Interval Graph Completion. internal research report CS-91-18, Brown University, Providence, Rhode Island, Mar. 1991.
2. P. Bergner, P. Dahl, D. Engebretsen, and M. O'Keefe. Spill Code Minimization via Interference Region Spilling. *ACM SIG-PLAN Notices*, 32(5):287–295, May 1997.
3. D. Bernstein, D. Q. Goldin, M. C. Golumbic, H. Krawczyk, Y. Mansour, I. Nahshon, and R. Y. Pinter. Spill Code Minimization Techniques for Optimizing Compilers. *SIGPLAN Notices*, 24(7):258–263, July 1989. Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation.
4. D. Berson, R. Gupta, and M. Soffa. URSA: A unified ReSource allocator for registers and functional units in VLIW architectures. In *Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, pages 243–254, Orlando, Florida, Jan. 1993.
5. D. A. Berson. *Unification of Register Allocation and Instruction Scheduling in Compilers for Fine-Grain Parallel Architecture*. PhD thesis, Pittsburgh University, 1996.
6. D. G. Bradlee, S. J. Eggers, and R. R. Henry. Integrating Register Allocation and Instruction Scheduling for RISCs. *ACM SIGPLAN Notices*, 26(4):122–131, Apr. 1991.
7. T. S. Brasier. FRIGG: A New Approach to Combining Register Assignment and Instruction Scheduling. Master thesis, Michigan Technological University, 1994.
8. D. Callahan and B. Koblenz. Register Allocation via Hierarchical Graph Coloring. *SIGPLAN Notices*, 26(6):192–203, June 1991. Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.
9. G. J. Chaitin. Register allocation and spilling via graph coloring. *ACM SIG-PLAN Notices*, 17(6):98–105, June 1982.
10. P. Crawley and R. P. Dilworth. *Algebraic Theory of Lattices*. Prentice Hall, Englewood Cliffs, 1973.
11. J. R. Goodman and W.-C. Hsu. Code Scheduling and Register Allocation in Large Basic Blocks. In *Conference Proceedings 1988 International Conference on Supercomputing*, pages 442–452, St. Malo, France, July 1988.
12. C. Norris and L. L. Pollock. A Scheduler-Sensitive Global Register Allocator. In IEEE, editor, *Supercomputing 93 Proceedings: Portland, Oregon*, pages 804–813, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, Nov. 1993. IEEE Computer Society Press.
13. S. S. Pinter. Register Allocation with Instruction Scheduling: A New Approach. *SIGPLAN Notices*, 28(6):248–257, June 1993.
14. M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, Sept. 1999.
15. Schlansker, B. Rau, and S. Mahlke. Achieving High Levels of instruction-Level Parallelism with Reduced Hardware Complexity. Technical Report HPL-96-120, Hewlet Packard, 1994.
16. S.-A.-A. Touati. Optimal Register Saturation in Acyclic Superscalar and VLIW Codes. Research Report, INRIA, Nov. 2000.
<ftp.inria.fr/INRIA/Projects/a3/touati/optiRS.ps.gz>.
17. S.-A.-A. Touati and F. Thomasset. Register Saturation in Data Dependence Graphs. Research Report RR-3978, INRIA, July 2000.
<ftp.inria.fr/INRIA/publication/publi-ps-gz/RR/RR-3978.ps.gz>.

Directly-Executable Earley Parsing

John Aycock and Nigel Horspool

Department of Computer Science,
University of Victoria,
Victoria, B. C., Canada V8W 3P6
{aycock,nigelh}@csc.uvic.ca

Abstract. Deterministic parsing techniques are typically used in favor of general parsing algorithms for efficiency reasons. However, general algorithms such as Earley’s method are more powerful and also easier for developers to use, because no seemingly arbitrary restrictions are placed on the grammar. We describe how to narrow the performance gap between general and deterministic parsers, constructing a directly-executable Earley parser that can reach speeds comparable to deterministic methods even on grammars for commonly-used programming languages.

1 Introduction

Most parsers in use today are only capable of handling subsets of context-free grammars: LL, LR, LALR. And with good reason – efficient linear-time algorithms for parsing these subsets are known. In contrast, general parsers which can handle any context-free grammar are slower due to extra overhead, even for those cases where the general algorithm runs in linear time [13].

However, general algorithms have some advantages. No “massaging” of a context-free grammar is required to make it acceptable for use in a general parser, as is required by more efficient algorithms like the LALR(1) algorithm used in Yacc [15]. Using a general parser thus reduces programmer development time, eliminates a source of potential bugs, and lets the grammar reflect the input language rather than the limitations of a compiler tool.

General algorithms also work for ambiguous grammars, unlike their more efficient counterparts. Some programming language grammars, such as those for Pascal, C, and C++, contain areas of ambiguity. For some tasks ambiguous grammars may be deliberately constructed, such as a grammar which describes multiple dialects of a language for use in software reengineering [7].

The primary objection to general parsing algorithms, then, is not one of functionality but of speed. For LR parsers, dramatic speed improvements have been obtained by producing hard-coded, or directly-executable parsers [3,5,14,23,24]. These directly-executable LR parsers implement the parser for a given grammar as a specialized program, rather than using a typical table-driven approach. In this paper we extend directly-executable techniques for use in Earley’s general

parsing algorithm, to produce and evaluate what we believe is the first directly-executable Earley parser. The speed of our parsers is shown to be comparable to deterministic parsers produced by Bison.

2 Earley Parsing

Earley’s algorithm [9,10] is a general parsing algorithm; it can recognize input described by any context-free grammar (CFG). (We assume the reader is familiar with the standard definition and notation of CFGs.) As in [11], uppercase letters (A, B) represent nonterminals, lowercase letters (a, b) represent terminals, and α and β are strings of terminal and nonterminal symbols. Also, every CFG G is augmented by adding a new start rule $S' \rightarrow S$, where S is the original start symbol of G .

Earley’s algorithm works by building a sequence of Earley sets¹ one initial Earley set S_0 , and one Earley set S_i for each input symbol x_i . An Earley set contains Earley items, which consist of three parts: a grammar rule; a position in the grammar rule’s right-hand side indicating how much of that rule has been seen, denoted by a dot (\bullet); a pointer back to some previous “parent” Earley set. For instance, the Earley item $[A \rightarrow a \bullet Bb, 12]$ indicates that the parser has seen the first symbol of the grammar rule $A \rightarrow aBb$, and points back to Earley set S_{12} . We use the term “core Earley item” to refer to an Earley item less its parent pointer: $A \rightarrow a \bullet Bb$ in the above example.

The three steps below are applied to Earley items in S_i until no more can be added; this constructs S_i and primes S_{i+1} .

SCANNER. If $[A \rightarrow \dots \bullet b \dots, j]$ is in S_i and $x_i = b$, add $[A \rightarrow \dots b \bullet \dots, j]$ to S_{i+1} .

PREDICTOR. If $[A \rightarrow \dots \bullet B \dots, j]$ is in S_i , add $[B \rightarrow \bullet \alpha, i]$ to S_i for all rules $B \rightarrow \alpha$ in G .

COMPLETER. If a “final” Earley item $[A \rightarrow \dots \bullet, j]$ is in S_i , add $[B \rightarrow \dots A \bullet \dots, k]$ to S_i for all Earley items $[B \rightarrow \dots \bullet A \dots, k]$ in S_j .

An Earley item is added to a Earley set only if it is not already present in the Earley set. The initial set S_0 holds the single Earley item $[S' \rightarrow \bullet S, 0]$ prior to Earley set construction, and the final Earley set must contain $[S' \rightarrow S \bullet, 0]$ upon completion in order for the input string to be accepted. For example, Fig. 1 shows the Earley sets when parsing an input using the expression grammar G_E :

$$\begin{array}{lll} S' \rightarrow E & T \rightarrow T * F & F \rightarrow n \\ E \rightarrow E + T & T \rightarrow T / F & F \rightarrow -F \\ E \rightarrow E - T & T \rightarrow F & F \rightarrow +F \\ E \rightarrow T & & F \rightarrow (E) \end{array}$$

¹ To avoid confusion later, we use the unfortunately awkward terms “Earley set” and “Earley item” throughout.

S_0	n	$+$	n
$S' \rightarrow \bullet E, 0$ $E \rightarrow \bullet E + T, 0$ $E \rightarrow \bullet E - T, 0$ $E \rightarrow \bullet T, 0$ $T \rightarrow \bullet T * F, 0$ $T \rightarrow \bullet T / F, 0$ $T \rightarrow \bullet F, 0$ $F \rightarrow \bullet n, 0$ $F \rightarrow \bullet - F, 0$ $F \rightarrow \bullet + F, 0$ $F \rightarrow \bullet (E), 0$	$F \rightarrow n \bullet, 0$ $T \rightarrow F \bullet, 0$ $E \rightarrow T \bullet, 0$ $T \rightarrow T \bullet * F, 0$ $T \rightarrow T \bullet / F, 0$ $S' \rightarrow E \bullet, 0$ $E \rightarrow E \bullet + T, 0$ $E \rightarrow E \bullet - T, 0$	$E \rightarrow E + \bullet T, 0$ $T \rightarrow \bullet T * F, 2$ $T \rightarrow \bullet T / F, 2$ $T \rightarrow \bullet F, 2$ $F \rightarrow \bullet n, 2$ $F \rightarrow \bullet - F, 2$ $F \rightarrow \bullet + F, 2$ $F \rightarrow \bullet (E), 2$	$F \rightarrow n \bullet, 2$ $T \rightarrow F \bullet, 2$ $E \rightarrow E + T \bullet, 0$ $T \rightarrow T \bullet * F, 2$ $T \rightarrow T \bullet / F, 2$ $S' \rightarrow E \bullet, 0$

Fig. 1. Earley sets for the expression grammar G_E , parsing the input $n + n$. Emboldened final Earley items are ones which correspond to the input's derivation.

The Earley algorithm may employ lookahead to reduce the number of Earley items in each Earley set, but we have found the version of the algorithm without lookahead suitable for our purposes. We also restrict our attention to input recognition rather than parsing proper. Construction of parse trees in Earley's algorithm is done after recognition is complete, based on information retained by the recognizer, so this division may be done without loss of generality.

There are a number of observations about Earley's algorithm which can be made. By themselves, they seem obvious, yet taken together they shape the construction of our directly-executable parser.

- Observation 1. Additions are only ever made to the current and next Earley sets, S_i and S_{i+1} .
- Observation 2. The COMPLETER does not recursively look back through Earley sets; it only considers a single parent Earley set, S_j .
- Observation 3. The SCANNER looks at each Earley item exactly once, and this is the only place where the dot may be moved due to a terminal symbol.
- Observation 4. Earley items added by PREDICTOR all have the same parent, i .

3 DEEP: A Directly-Executable Earley Parser

3.1 Basic Organization

The contents of an Earley set depend on the input and are not known until run time; we cannot realistically precompute one piece of directly-executable code for every possible Earley set. We can assume, however, that the grammar is known prior to run time, so we begin by considering how to generate one piece of directly-executable code per Earley item.

Even within an Earley item, not everything can be precomputed. In particular, the value of the parent pointer will not be known ahead of time. Given two

otherwise identical Earley items $[A \rightarrow \alpha \bullet \beta, j]$ and $[A \rightarrow \alpha \bullet \beta, k]$, the invariant part is the core Earley item. The code for a directly-executable Earley item, then, is actually code for the core Earley item; the parent pointer is maintained as data. A directly-executable Earley item may be represented as the tuple

$$(code \text{ for } A \rightarrow \alpha \bullet \beta, parent)$$

the code for which is structured as shown in Fig. 2. Each terminal and non-terminal symbol is represented by a distinct number; the variable `sym` can thus contain either type of symbol. Movement over a terminal symbol is a straightforward implementation of the SCANNER step, but movement over a nonterminal is complicated by the fact that there are two actions that may take place upon reaching an Earley item $[A \rightarrow \dots \bullet B \dots, j]$, depending on the context:

1. If encountered when processing the current Earley set, the PREDICTOR step should be run.
2. If encountered in a parent Earley set (i.e., the COMPLETER step is running) then movement over the nonterminal may occur. In this case, `sym` cannot be a terminal symbol, so the predicate `ISTERMINAL()` is used to distinguish these two cases.

The code for final Earley items calls the code implementing the parent Earley set, after replacing `sym` with the nonterminal symbol to move the dot over. By Observation 2, no stack is required as the call depth is limited to one. Again, this should only be executed if the current set is being processed, necessitating the `ISTERMINAL()`.

3.2 Earley Set Representation

An Earley set in the directly-executable representation is conceptually an ordered sequence of $(code, parent)$ tuples followed by one of the special tuples:

$$\begin{aligned} &(end \text{ of current Earley set } code, -1) \\ &(end \text{ of parent Earley set } code, -1) \end{aligned}$$

The code at the end of a parent Earley set simply contains a `return` to match the `call` made by a final Earley item. Reaching the end of the current Earley set is complicated by bookkeeping operations to prepare for processing the next Earley set. The parent pointer is irrelevant for either of these.

In practice, our DEEP implementation splits the tuples. DEEP's Earley sets are in two parts: a list of addresses of code, and a corresponding list of parent pointers. The two parts are separated in memory by a constant amount, so that knowing the memory location of one half of a tuple makes it a trivial calculation to find the other half.

Having a list of code addresses for an Earley set makes it possible to implement the action “`goto next Earley item`” with direct threaded code [4]. With threaded code, each directly-executable Earley item jumps directly to the beginning of the code for the next Earley item, rather than first returning to

$[A \rightarrow \dots \bullet a \dots, j]$ (movement over a terminal)	\Rightarrow	<pre> if (sym == a) { add [A → ⋯ a • ⋯, j] to S_{i+1} } goto next Earley item </pre>
$[A \rightarrow \dots \bullet B \dots, j]$ (movement over a nonterminal)	\Rightarrow	<pre> if (ISTERMINAL(sym)) { foreach rule B → α { add [B → • α, i] to S_i } } else if (sym == B) { add [A → ⋯ B • ⋯, j] to S_i } goto next Earley item </pre>
$[A \rightarrow \dots \bullet, j]$ (final Earley item)	\Rightarrow	<pre> if (ISTERMINAL(sym)) { saved_sym = sym sym = A call code for Earley set S_j sym = saved_sym } goto next Earley item </pre>

Fig. 2. Pseudocode for directly-executable Earley items.

a dispatching loop or incurring function call overhead. Not only does threaded code proffer speed advantages [16], but it can work better with branch prediction hardware on modern CPUs [11]. We implement this in a reasonably portable fashion using the first-class labels in GNU C.²

How is an Earley item added to an Earley set in this representation? First, recall that an Earley item is only placed in an Earley set if it does not already appear there. We will use the term “appending” to denote an Earley item being placed into an Earley set; “adding” is the process of determining if an Earley item should be appended to an Earley set. (We discuss adding more in Section 3.3.) Using this terminology, appending an Earley item to an Earley set is done by dynamically generating threaded code. We also dynamically modify the threaded code to exchange one piece of code for another in two instances:

1. When the current Earley set is fully processed, “end of current Earley set” must be exchanged for “end of parent Earley set.”
2. Observation 3 implies that once the SCANNER has looked at an Earley item, any code looking for terminal symbols is superfluous. By modifying the threaded code, DEEP skips the superfluous code on subsequent invocations.

Appending leaves DEEP with a thorny problem of memory management. Observation 1 says that Earley items – threaded code plus separate parent pointers – can be appended to one of two Earley sets. We also maintain an array whose

² This is an extension to ANSI C.

i^{th} entry is a pointer to the code for Earley set S_i , for implementation of `call`, giving us a total of five distinct, dynamically-growing memory areas.

Instead of complex, high-overhead memory management, we have the operating system assist us by memory-mapping oversized areas of virtual memory. This is an efficient operation because the operating system will not allocate the virtual memory pages until they are used. We can also protect key memory pages so that an exception is caused if DEEP should exceed its allocated memory, absolving us from performing bounds checking when appending. This arrangement is shown in Fig. 3, which also demonstrates how the current and next Earley sets alternate between memory areas.

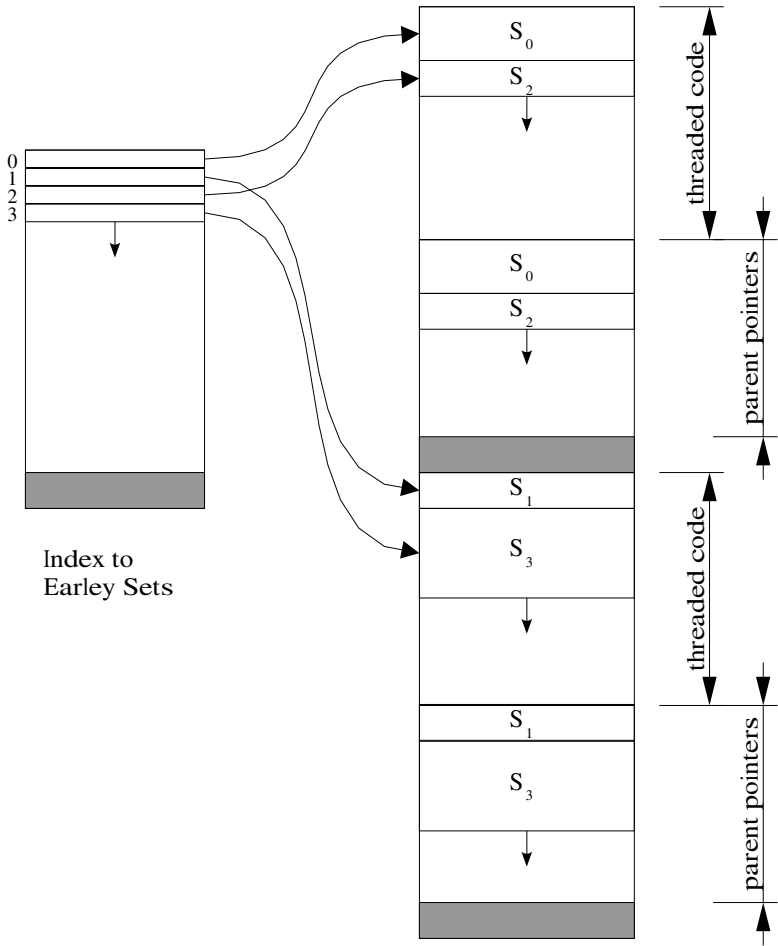


Fig. 3. Memory layout for DEEP. S_2 and S_3 are the current and next Earley sets, respectively; the shaded areas are protected memory pages.

3.3 Adding Earley Items

As mentioned, adding an Earley item to an Earley set entails checking to ensure that it is not already present. Earley suggested using an array indexed by the parent pointer, each entry of which would be a list of Earley items to search [10]. Instead, we note that core Earley items may be enumerated, yielding finite, relatively small numbers [3]. A core Earley item's number may be used to index into a bitmap to quickly check the presence or absence of *any* Earley item with that core.

When two or more Earley items exist with the same core, but different parent pointers, we construct a radix tree [17] for that core Earley item – a binary tree whose branches are either zero or one – which keeps track of which parent pointers have been seen. Radix trees have two nice properties:

1. Insertion and lookup, the only operations required, are simple.
2. The time complexity of radix tree operations during execution is $\log i$, where i is the number of tokens read, thus growing slowly even with large inputs.

To avoid building a radix tree for a core Earley item until absolutely necessary, we cache the first parent pointer until we encounter a second Earley item with the same core. An example of adding Earley items is given in Fig. 4.

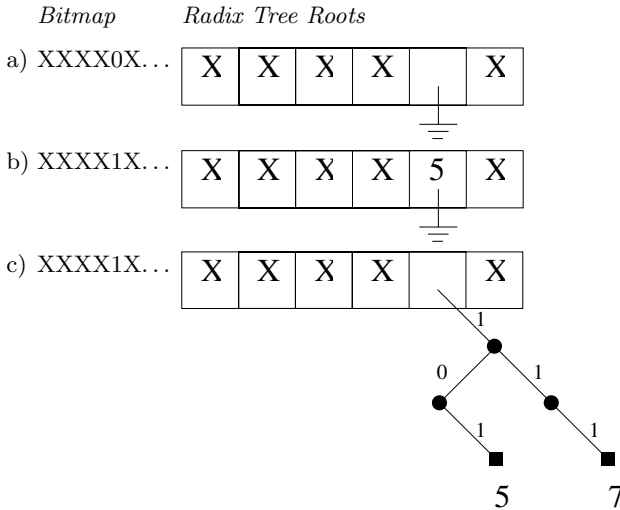


Fig. 4. Adding Earley items: (a) initial state; (b) after appending Earley item #4, parent S_5 ; (c) after appending Earley item #4, parent S_7 . In the radix tree, a circle (●) indicates an absent entry, and a square (■) indicates an existing entry. “X” denotes a “don’t care” entry.

³ To be precise, the number of core Earley items is $\sum_{A \rightarrow \alpha \in G} (|\alpha| + 1)$.

together results in a complex, inelegant implementation. However, we realized as a result of Observation 4 that the PREDICTOR really just corresponds to making a transition to a “nonkernel” state in the LR(0) DFA. Pursuing this idea, we represent Earley items in DEEP as the tuples

(code for LR(0) DFA state, parent)

Figure 6 shows the Earley sets from Fig. 1 recoded using the LR(0) DFA states.

	n		+		n	
S_0		S_1		S_2		S_3
0 , 0 1 , 0		3 , 0 8 , 0 9 , 0 10 , 0 2 , 0		19 , 0 18 , 2		3 , 2 8 , 2 24 , 2 25 , 0 2 , 0

Fig. 6. Earley sets for the expression grammar G_E , parsing the input $n + n$, encoded using LR(0) DFA states.

Through this new representation, we gain most of the efficiency of using an LR(0) DFA as the basis of an Earley parser, but with the benefit of a particularly simple representation and implementation. The prior discussion in this section regarding DEEP still holds, except the directly-executable code makes transitions from one LR(0) DFA state to another instead of from one Earley item to another.

4 Evaluation

We compared DEEP with three different parsers:

1. ACCENT, an Earley parser generator [25].
2. A standard implementation of an Earley parser, by the second author.
3. Bison, the GNU incarnation of Yacc.

All parsers were implemented in C, used the same (flex-generated) scanner, and were compiled with gcc version 2.7.2.3 using the -O flag. Timings were conducted on a 200 MHz Pentium with 64 M of RAM running Debian GNU/Linux version 2.1.

Figure 7 shows the performance of all four on G_E , the expression grammar from Sect. 2. As expected, Bison is the fastest, but DEEP is a close second, markedly faster than the other Earley parsers.

In Fig. 8 the parsers (less Bison) operate on an extremely ambiguous grammar. Again, DEEP is far faster than the other Earley parsers. The performance curves themselves are typical of the Earley algorithm, whose time complexity is $O(n)$ for most LR(k) grammars, $O(n^2)$ for unambiguous grammars, and $O(n^3)$ in the worst case. [10]

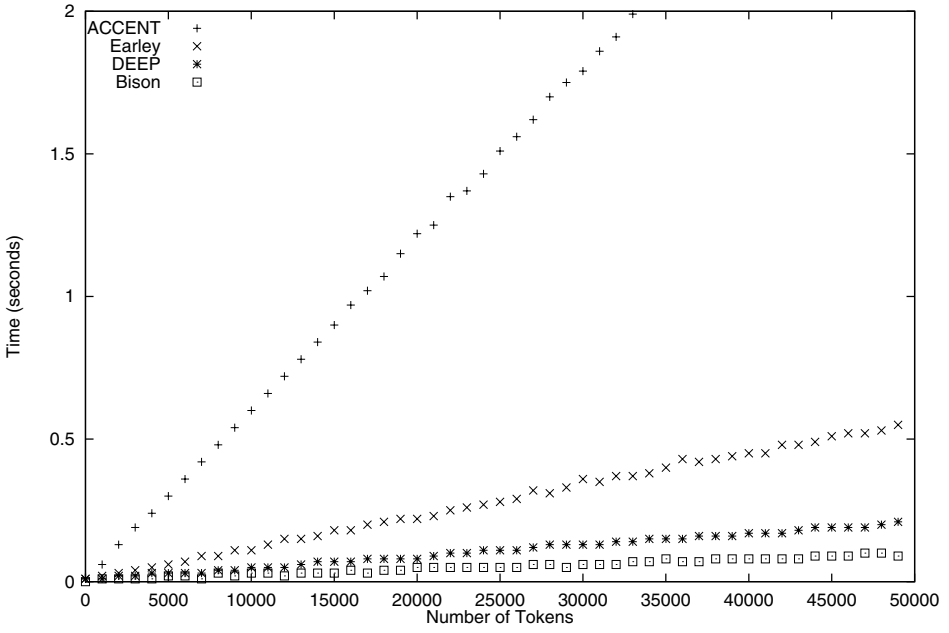


Fig. 7. Timings for the expression grammar, G_E .

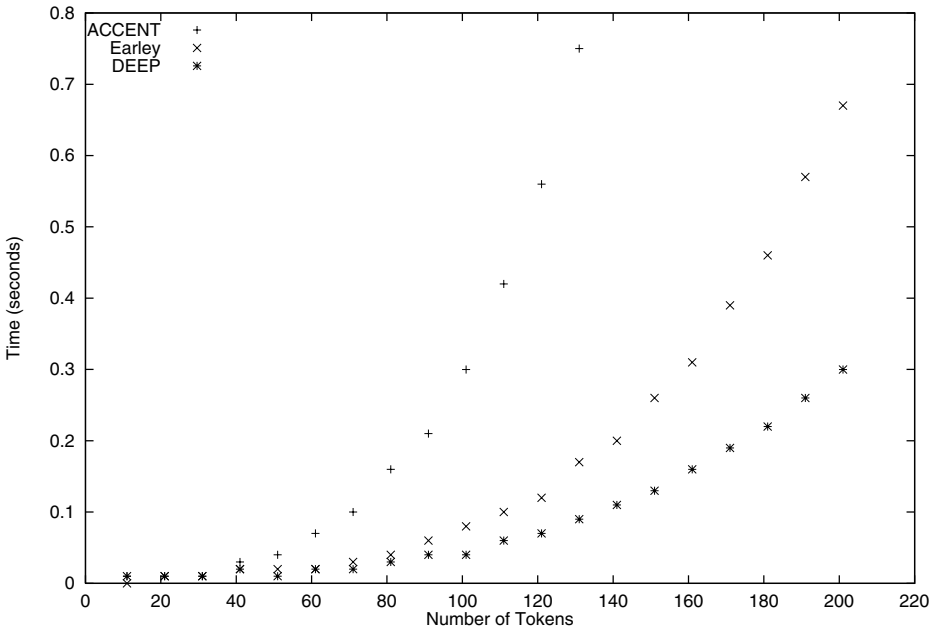


Fig. 8. Timings for the ambiguous grammar $S \rightarrow SSx|x$.

5 Improvements

Next, we tried DEEP on the Java 1.1 grammar [12] which consists of 350 grammar rules⁴. Suffice it to say that only the extremely patient would be content to wait while `gcc` compiled and optimized this monster. To make DEEP practical, its code size had to be reduced.

Applying Observation 3, code looking for terminal symbols may only be executed once during the lifetime of a given Earley item. In contrast, an Earley item’s nonterminal code may be invoked many times. To decrease the code size, we excised the directly-executable code for terminal symbols, replacing it with a single table-driven interpreter which interprets the threaded code. Nonterminal code is still directly-executed, when COMPLETER calls back to a parent Earley set. This change reduced compile time by over 90%.

Interpretation allowed us to trivially make another improvement, which we call “Earley set compression.” Often, states in the $LR(0)$ \overline{DFA} have no transitions on nonterminal symbols; the corresponding directly-executable code is a no-op which can consume both time and space. The interpreter looks for such cases and removes those Earley items, since they cannot contribute to the parse. We think of Earley set compression as a space optimization, because only a negligible performance improvement resulted from its implementation.

The version of DEEP using partial interpretation and Earley set compression is called SHALLOW. Figure 9 compares the performance of SHALLOW and Bison on realistic Java inputs. SHALLOW can be two to five times slower, although the difference amounts to only fractions of a second – a difference unlikely to be noticed by end users!

One typical improvement to Earley parsers is the use of lookahead. Earley suggested that the COMPLETER should employ lookahead, but this was later shown to be a poor choice [8]. Instead, it was demonstrated that the use of one-token lookahead by the PREDICTOR yielded the best results [8]. This “prediction lookahead” avoids placing Earley items into an Earley set that are obvious dead ends, given the subsequent input. However, the $LR(0)$ \overline{DFA} naturally clusters together PREDICTOR’s output. Where prediction lookahead would avoid generating many Earley items in a standard Earley parser, it would only avoid one Earley item in SHALLOW if all the predicted dead ends fell within a single $LR(0)$ \overline{DFA} state.

We instrumented SHALLOW to track Earley items that were unused, in the sense that they never caused more Earley items to be added to any Earley set, and were not final Earley items. Averaging over the Java corpus, 16% of the Earley items were unused. Of those, prediction lookahead could remove *at most* 19%; Earley set compression removed 76% of unused Earley items in addition to pruning away other Earley items. We conjecture that prediction lookahead is of limited usefulness in an Earley parser using any type of LR automaton.

⁴ This number refers to grammar rules in Backus-Naur form, obtained after transforming the grammar from [12] in the manner they prescribe.

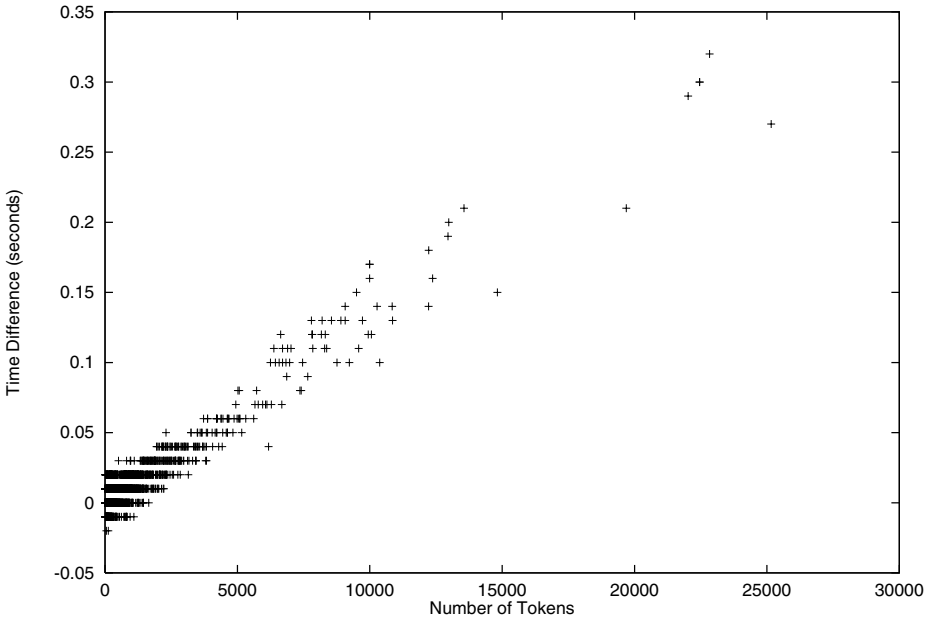


Fig. 9. Difference between SHALLOW and Bison timings for Java 1.1 grammar, parsing 3350 Java source files from JDK 1.2.2 and Java Cup v10j.

6 Related Work

Appropriately, the first attempt at direct execution of an Earley parser was made by Earley himself [9]. For a subset of the CFGs which his algorithm recognized in linear time, he proposed an algorithm to produce a hardcoded parser. Assuming the algorithm worked and scaled to practically-sized grammars – Earley never implemented it – it would only work for a subset of CFGs, and it possessed unresolved issues with termination.

The only other reference to a directly-executable “Earley” parser we have found is Leermakers’ recursive ascent Earley parser [19,20,21]. He provides a heavily-recursive functional formulation which, like Earley’s proposal, appears not to have been implemented. Leermakers argues that the directly-executable LR parsers which influenced our work are really just manifestations of recursive ascent parsers [20], but he also notes that he uses “recursive ascent Earley parser” to denote parsers which are not strictly Earley ones [21, page 147]. Indeed, his algorithm suffers from a problem handling cyclic grammar rules, a problem not present in Earley’s algorithm (and consequently not present in our Earley parsers).

Using deterministic parsers as an efficient basis for general parsing algorithms was suggested by Lang in 1974 [18]. However, none of the applications of this idea in Earley parsers [22] and Earley-like parsers [26,26] have explored the benefits

of using an almost-deterministic automaton and exploiting Earley’s ability to simulate nondeterminism.

7 Future Work

By going from DEEP to SHALLOW, we arrived at a parser suited for practical use. This came at a cost, however: as shown in Fig. 10, the partially-interpreted SHALLOW is noticeably slower than the fully-executable DEEP. Even with the slowdown, SHALLOW’s timings are still comparable to Bison’s. One area of future work is determining how to retain DEEP’s speed while maintaining the practicality of SHALLOW.

On the other side of the time/space coin, we have yet to investigate ways to reduce the size of generated parsers. Comparing the sizes of stripped executables, SHALLOW parsers for G_E and Java were 1.5 and 9.0 times larger, respectively, than the corresponding Bison parsers.

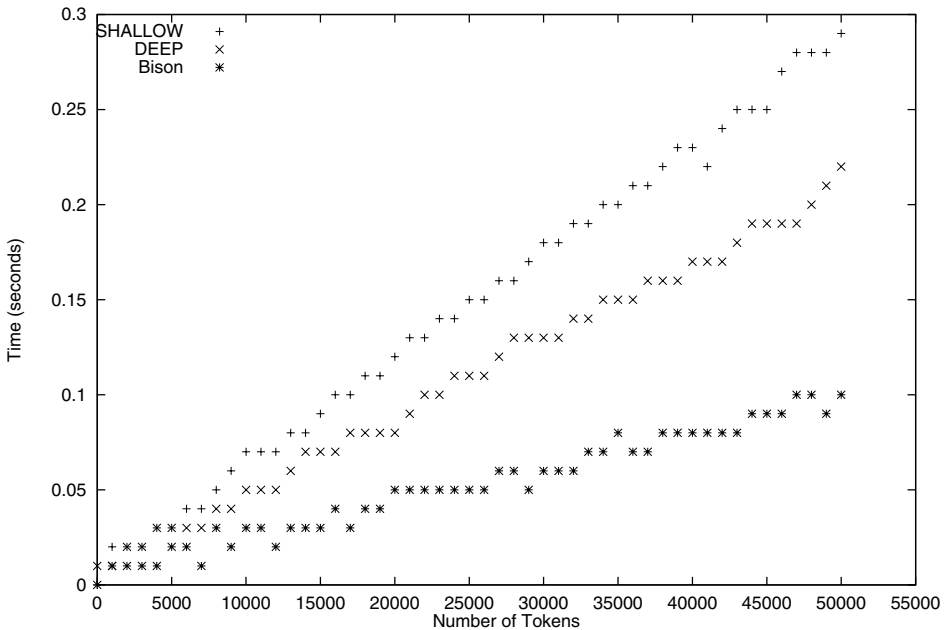


Fig. 10. Performance impact of partial interpretation of G_E .

Additionally, we have not yet explored the possibility of using optimizations based on grammar structure. One such example is elimination of unit rules⁵

⁵ Also called chain rule elimination.

grammar rules such as $A \rightarrow B$ with only a single nonterminal on the right-hand side [13]. Techniques like this have been employed with success in other directly-executable parsers [14,24].

8 Conclusion

We have shown that directly-executable LR parsing techniques can be extended for use in general parsing algorithms such as Earley's algorithm. The result is a directly-executable Earley parser which is substantially faster than standard Earley parsers, to the point where it is comparable with LALR(1) parsers produced by Bison.

Acknowledgments. This work was supported in part by a grant from the National Science and Engineering Research Council of Canada. Shannon Jaeger made a number of helpful comments on a draft of this paper.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
2. M. A. Alonso, D. Cabrero, and M. Vilares. Construction of Efficient Generalized LR Parsers. *Proceedings of the Second International Workshop on Implementing Automata*, 1997, pp. 131–140.
3. D. T. Barnard and J. R. Cordy. SL Parses the LR Languages. *Computer Languages* 13, 2 (1988), pp. 65–74.
4. J. R. Bell. Threaded Code. *CACM* 16, 6 (June 1973), pp. 370–372.
5. A. Bhamidipaty and T. A. Proebsting. Very Fast YACC-Compatible Parsers (For Very Little Effort). *Software: Practice and Experience* 28, 2 (February 1998), pp. 181–190.
6. S. Billot and B. Lang. The Structure of Shared Forests in Ambiguous Parsing. *Proceedings of the 27th Annual Meeting of the Association for Computational Linguistics*, 1989, pp. 143–151.
7. M. van den Brand, A. Sellink, and C. Verhoef. Current Parsing Techniques in Software Renovation Considered Harmful. *International Workshop on Program Comprehension*, 1998, pp. 108–117.
8. M. Bouckaert, A. Pirotte, and M. Snelling. Efficient Parsing Algorithms for General Context-free Parsers. *Information Sciences* 8, 1975, pp. 1–26.
9. J. Earley. *An Efficient Context-Free Parsing Algorithm*, Ph.D. thesis, Carnegie-Mellon University, 1968.
10. J. Earley. An Efficient Context-Free Parsing Algorithm. *CACM* 13, 2 (February 1970), pp. 94–102.
11. M. A. Ertl and D. Gregg. Hardware Support for Efficient Interpreters: Fast Indirect Branches (Draft). May, 2000.
12. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.
13. D. Grune and C. J. H. Jacobs. *Parsing Techniques: A Practical Guide*. Ellis Horwood, 1990.

14. R. N. Horspool and M. Whitney. Even Faster LR Parsing. *Software: Practice and Experience* 20, 6 (June 1990), pp. 515–535.
15. S. C. Johnson. Yacc: Yet Another Compiler-Compiler. *Unix Programmer's Manual* (7th edition), volume 2B, 1978.
16. P. Klint. Interpretation Techniques. *Software: Practice and Experience* 11, 1981, pp. 963–973.
17. D. E. Knuth. *The Art of Computer Programming Volume 3: Sorting and Searching* (2nd edition), Addison-Wesley, 1998.
18. B. Lang. Deterministic Techniques for Efficient Non-Deterministic Parsers. In *Automata, Languages, and Programming (LNCS #14)*, J. Loeckx, ed., Springer-Verlag, 1974.
19. R. Leermakers. A recursive ascent Earley parser. *Information Processing Letters* 41, 1992, pp. 87–91.
20. R. Leermakers. Recursive ascent parsing: from Earley to Marcus. *Theoretical Computer Science* 104, 1992, pp. 299–312.
21. R. Leermakers. *The Functional Treatment of Parsing*. Kluwer Academic, 1993.
22. P. McLean and R. N. Horspool. A Faster Earley Parser. *Proceedings of the International Conference on Compiler Construction, CC '96*, 1996, pp. 281–293.
23. T. J. Pennello. Very Fast LR Parsing. *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction, SIGPLAN 21*, 7 (1986), pp. 145–151.
24. P. Pfahler. Optimizing Directly Executable LR Parsers. *Compiler Compilers, Third International Workshop, CC '90*, 1990, pp. 179–192.
25. F. W. Schröer. *The ACCENT Compiler Compiler, Introduction and Reference*. GMD Report 101, German National Research Center for Information Technology, June 2000.
26. M. Vilares Ferro and B. A. Dion. Efficient Incremental Parsing for Context-Free Languages. *Proceedings of the 5th IEEE International Conference on Computer Languages*, 1994, pp. 241–252.

A Bounded Graph-Connect Construction for LR-regular Parsers

Jacques Farré¹ and José Fortes Gálvez^{2,1}

¹ Laboratoire I3S, CNRS and Université de Nice - Sophia Antipolis

² Depart. de Informática y Sistemas, Universidad de Las Palmas de Gran Canaria

Abstract. Parser generation tools currently used for computer language analysis rely on user wisdom in order to resolve grammar conflicts. Here practical LR(0)-based parser generation is introduced, with automatic conflict resolution by potentially-unbounded lookahead exploration. The underlying LR(0)-automaton item dependence graph is used for lookahead DFA construction. A bounded graph-connect technique overcomes the difficulties of previous approaches with empty rules, and compact coding allows to precisely resume right-hand contexts. Resulting parsers are deterministic and linear, and accept a large class of LR-regular grammars including LALR(k). Their construction is formally introduced, shown to be decidable, and illustrated by a detailed example.

1 Introduction

Grammars for many computer languages (programming languages, description languages ...) are neither LL(1) nor LALR(1). Unfortunately, most available parser generators, without user help, are restricted to these grammar classes. And the user is confronted with adapting the grammar, if not the language, to the restrictions imposed by the tool. Unnatural design, which may not adapt well to the application, results from these restrictions, e.g., the discussion on a Java LALR grammar in [8]. Moreover, this adaptation often requires user expertise and detailed analysis. The need for more powerful parsers exists, as shown by the popularity of tools proposing different means to overcome conflicts. A first choice is to use generators that propose to resolve conflicts by means of predicates [9, 14]. These are apparently-simple semantic checks or lookahead explorations specified “ad hoc” by the user. But resorting to predicates is intrinsically insecure, because parser generators cannot detect design errors, and may produce incorrect parsers without user notice. Moreover, ill-designed syntactic predicates may involve heavy backtracking, resulting in inefficient parsers.

A second choice is to use parsers for unrestricted context-free grammars [6, 12, 11]. In particular *Generalized LR* [18], which has efficient implementations [1] and has been successfully used outside the scope of natural language processing (e.g., [19]). Unfortunately, these generators cannot always warn about grammar ambiguity. As a result, the user may be confronted with an unexpected forest at parsing time. Although GLR has been practically found near-linear in many

cases, nondeterminism occurs for each conflict, and linearity is not guaranteed, even for unambiguous grammars. Finally, nondeterminism compels to undo or to defer semantic actions.

We think that, between the above choices, a tool offering automatic conflict resolution through potentially-unbounded lookahead exploration should be useful. It would allow to produce deterministic and reliable parsers for a large class of unambiguous grammars, without relying on user help or expertise.

In LR-regular (LRR) parsing [5], an LR conflict can be resolved if there exist disjoint regular languages covering the right-hand contexts of every action in conflict. The LRR class is theoretically interesting, since it allows to build efficient parsers using deterministic finite-state automata (DFA) for lookahead exploration. In particular, it has been established that LRR parsing is linear [10], what is not ensured by GLR or parsers with predicates.

Unfortunately, full LRR parser generators cannot be constructed because it is undecidable whether a grammar is LRR [17]. Nevertheless, several attempts have been done to develop techniques for some subset of LRR, but have failed to produce sufficiently practical parser generators. At most, they guarantee LALR(k) only for grammars without ε -rules, what nowadays is considered unacceptable.

In this paper, we propose a bounded graph-connect technique for building LRR parsers. This technique is inspired by noncanonical discriminating-reverse (NDR) parsers [7], which actually accept a wider class of grammars not restricted within LRR. However, the method shown here naturally produces canonical and correct-prefix parsers (what is not ensured by NDR), which are usually considered desirable properties.

1.1 Related Work

The first attempt to develop parsers for a subset of LRR grammars is due to Baker [2], with the XLR method. In XLR, the LR automaton is used as the basis for the lookahead automata construction, with the addition of ε -arcs that allow to resume exploration after reduction states. However, the lack of precise right-hand context recovery makes XLR(k) a proper subclass of LALR(k).

The methods R(s)LR by Boullier [4], and LAR(s) by Bermudez and Schimpf [3], were independently developed with similar results. Their approach can be described as using a bounded memory of s LR(0) automaton states in order to improve right-hand context recovery. Thus, both methods are more powerful than XLR, but they both have problems with ε productions. As they show, R(s)LR(0) accepts all ε -free LALR(s) grammars, but for any s there exist LALR(1) grammars that are not LAR(s).

Seité studied automatic LALR(1) conflict resolution [15]. His method, while not more powerful than Boullier's in the lookahead exploration phase, introduces an additional phase which scans some stack prefix when the lookahead exploration has not been able to resolve the conflict. This extension, which increases the accepted class with some non-LR, LRR grammars, is quite independent of the lookahead exploration technique used. Although it can be applied to our method, we shall not consider it here.

1.2 Introduction to Our Approach

In our approach, right-hand context computation is made much more precise than previous methods, thanks to several innovations:

- Instead of automaton state transitions, we follow the underlying kernel-item transition graph of the LR(0) automaton. Using kernel items instead of full state item-set allows a simpler and more efficient construction.
- Instead of keeping track of s states for the right-hand context to recover, we keep track of at most h *path subgraphs* of the underlying item graph. Each such subgraph is coded by its extreme items, and the underlying graph is in turn used to follow precisely each possible path.
- An ε -skip move, combined with the above bounded graph-connect technique, allows to skip and precisely resume exploration on the right of ε -deriving nonterminals.

Our construction follows precisely right-hand contexts as far as no more than h path subgraphs are needed. Since at most a new subgraph is added for each explored terminal, all LALR(h) grammars are accepted. Moreover, for most practical grammars, an effective lookahead length well beyond h (in fact, usually unbounded, and thus allowing grammars not in LR) should be actually obtained.

In previous methods, since transitions on ε -deriving nonterminals add states to the (bounded) state memory, there exist LALR(1) grammars whose conflicts cannot be resolved.

2 Context Recovery and Subgraph Connections

In this section, we shall first describe the kernel-item underlying graph, and then the different aspects of our approach for computing right-hand contexts.

Usual notational conventions, e.g., [16], will be generally followed. The original grammar \mathcal{G} is augmented with $P' = \{S' \rightarrow S\} \cup P$. Productions in P' are numbered from 1 to $|P'|$, what is sometimes noted as $A \xrightarrow{i} \alpha$.

2.1 LR(0) Construction

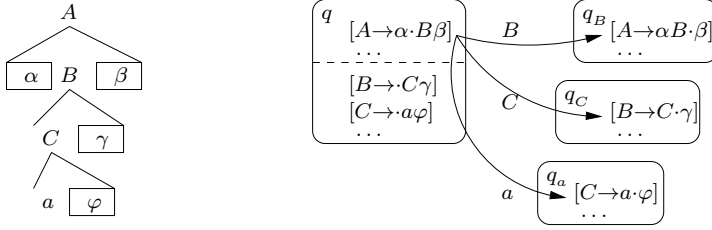
Let us first recall the LR(0) automaton construction where each state q corresponds to a set K_q of *kernel* items. For initial state q_0 we have $K_{q_0} = \{[S' \rightarrow \cdot S]\}$. The transition function Δ can be defined as follows:

$$\Delta(K_q, X) = \{[A \rightarrow \alpha X \cdot \beta] \mid [A \rightarrow \alpha \cdot X \beta] \in \mathcal{C}(K_q)\},$$

where \mathcal{C} is the *closure* operation, which can be defined as

$$\mathcal{C}(K) = K \cup \{[B \rightarrow \cdot \gamma] \mid [A \rightarrow \alpha \cdot \beta] \in K, \beta \xRightarrow{*}_{\text{rm}} Bx \Rightarrow \gamma x\}.$$

A kernel item ι indicates the parsing actions “shift” if there is some $[A \rightarrow \alpha \cdot a \beta]$ in $\mathcal{C}(\{\iota\})$ and “reduce j ” for each $[B \xrightarrow{j} \gamma \cdot]$ in $\mathcal{C}(\{\iota\})$. A *conflict*-state item set indicates a set of two or more different actions. For these states, our method builds a *lookahead* automaton, by using the kernel-item graph as its basic reference.

Fig. 1. Illustration of δ function

2.2 The Underlying LR(0) Kernel-Item Graph

In the LR(0) automaton underlying graph, each node $\nu = [A \rightarrow \alpha \cdot \beta]_q$ corresponds to some kernel item $[A \rightarrow \alpha \cdot \beta]$ in some set K_q , i.e., the same item in different state sets will correspond to different nodes. (Single) transitions are defined on V , as illustrated by Fig. 1.

$$\begin{aligned} \delta(\nu, X) = & \{ [A \rightarrow \alpha X \cdot \beta]_{q'} \mid \Delta(K_q, X) = K_{q'}, \nu = [A \rightarrow \alpha \cdot X \beta]_q \} \\ & \cup \{ [C \rightarrow X \cdot \gamma]_{q'} \mid \Delta(K_q, X) = K_{q'}, \nu = [A \rightarrow \alpha \cdot B \beta]_q, B \xrightarrow{\text{rm}}^* Cx \Rightarrow X\gamma x \}. \end{aligned}$$

In the first subset, the dot moves along the right-hand side of a rule until its right-hand end, where no more transitions are possible. The second subset may be understood as “transitions through” nonkernel items resulting by closure from some kernel item in the originating state q to some kernel item in the destination state q' .

We extend δ for transitions on strings in V^* :

$$\delta^*(\nu, \varepsilon) = \{\nu\}, \quad \delta^*(\nu, X\alpha) = \bigcup_{\nu' \in \delta(\nu, X)} \delta^*(\nu', \alpha).$$

We shall usually write $\nu \xrightarrow{\alpha} \nu'$ instead of $\nu' \in \delta^*(\nu, \alpha)$.

2.3 Simple Context Recovery

Item graph transitions on terminals determine the lookahead DFA transitions. The starting points, for each LR(0) conflict, are kernel items in the conflict state. Suppose that, in Fig. 2, q is a state with a shift-reduce conflict. For the shift action in conflict, the transition on a will be followed initially from node $[A \rightarrow \alpha \cdot B \beta]_q$ to node $[C \rightarrow a \cdot w]_{q_a}$, and then, if necessary, successive transitions on symbols in w .

A first aspect to consider is how to precisely recover the right-hand contexts of a left-hand side nonterminal when its corresponding rule's right-hand side has been completely traversed, e.g., in Fig. 2 when $\nu = [C \rightarrow a w \cdot]_{q_{aw}}$ is reached, graph traversal must resume with $\nu' = [D \rightarrow C \cdot \gamma]_{q_C}$, but not with nodes for which there exist transitions on C from some ν'' in $q' \neq q$. For this purpose,

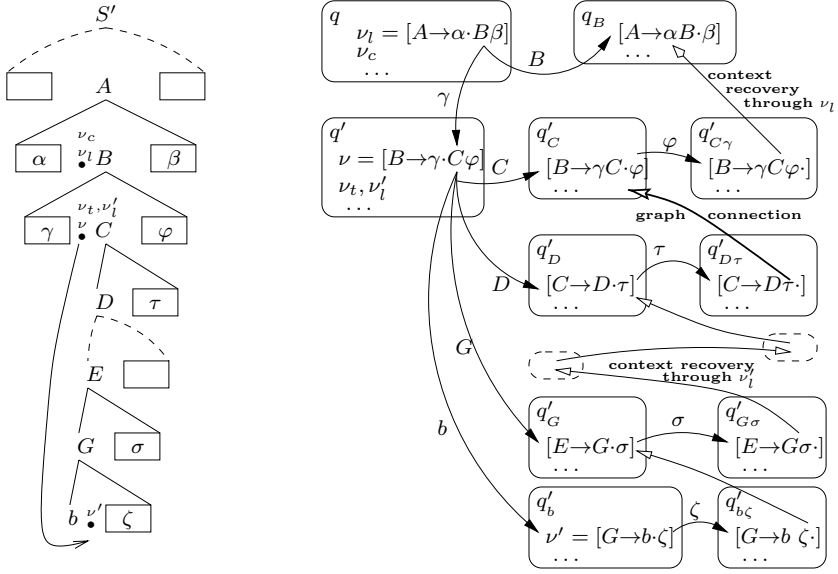


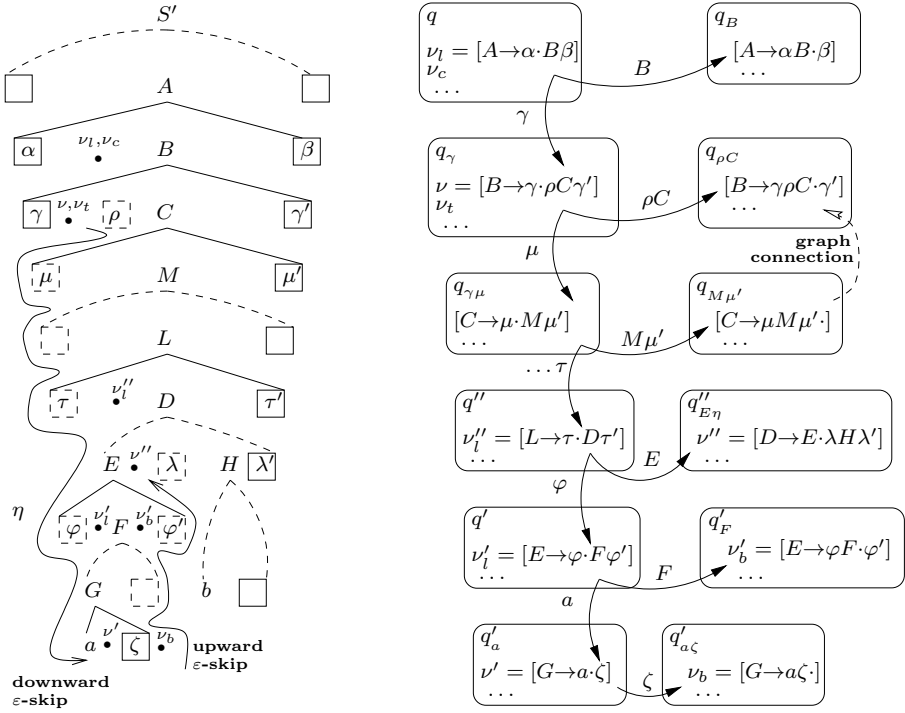
Fig. 3. Illustration of subgraph connection

appending another pair to a sequence of length h , the sequence's first element is lost. Thus, exact right-hand context cannot be computed if the lookahead automaton construction requires to resume graph transitions involving lost reference nodes.

2.5 Skipping ε -Deriving Nonterminals

Let us finally consider ε -productions. Differently from previous methods, the item graph allows to follow transitions on ε -deriving nonterminals without lengthening the graph sequence κ . If current position is $[B \rightarrow \gamma \cdot \rho C \gamma']_{q_\gamma}$, as in Fig. 4 —dashed boxes represent sequences of ε -deriving nonterminals—, and next exploration symbol can be $a \in \text{First}(C)$, a graph connection is performed after skipping the sequence η of ε -deriving nonterminals, and the pair (ν'_l, ν') is added to κ , with $\nu_t \xrightarrow{\eta} \nu'_l \xrightarrow{a} \nu'$. Thus, in general, $\nu_t^{(i)} \xrightarrow{\eta} \nu_c^{(i+1)}$, $\eta \Rightarrow^* \varepsilon$.

Context is recovered in principle as described in previous sections. However, we also need in general to skip sequences of ε -deriving nonterminals for transitions corresponding to up-right moves in the possible derivation trees, in order to reach a position on the left of some X such that $X \xRightarrow{\text{rm}}^* bx$, as shown by the upward ε -skip path in Fig. 4. For this purpose, amongst all paths allowed by the LR(0)-automaton graph, we should only consider those followed by the downward ε -skip from ν_t , i.e., those leading from ν_t to the current left-hand reference (ν'_l in Fig. 4).

Fig. 4. Illustration of skipping ε -deriving nonterminals

2.6 General Context Recovery

Let us now summarize the above considerations in order to present the general case for single (upward) context-recovery steps from current sequence $\kappa(\nu_l, \nu)$.

In general, we have $\nu_l \xrightarrow{\beta} \nu = [A \rightarrow \alpha \beta \cdot \gamma]_q$. If $\gamma \Rightarrow^* \varepsilon$, we have to consider an upward step to resume exploration for the (precise, if possible) right-hand context of A , i.e., we have to compute those sequences $\kappa'(\nu'_l, \nu')$ such that $\nu'_l \xrightarrow{\varphi A} \nu' = [B \rightarrow \varphi A \cdot \psi]_{q'}$ and which are compatible with current sequence. We distinguish the following cases:

1. ν' is lower than ν_l in the derivation tree² (leftmost tree in Fig. 5). Only ν changes, i.e., $\nu'_l = \nu_l$.
2. Otherwise, there are two subcases:
 - a) κ is empty, i.e., current sequence contains only one pair (ν_l, ν) (middle tree³ in Fig. 5 with no ν_t). If no graph connection has been performed yet,

² In this case, $\alpha = \varepsilon$.

³ The possible positions of the different nodes are shown. For instance, in this tree, lower ν_l passes to higher ν'_l when $\varphi = \varepsilon$, and higher ν_l —which is only possible when $\varphi \neq \varepsilon$ — does not move.

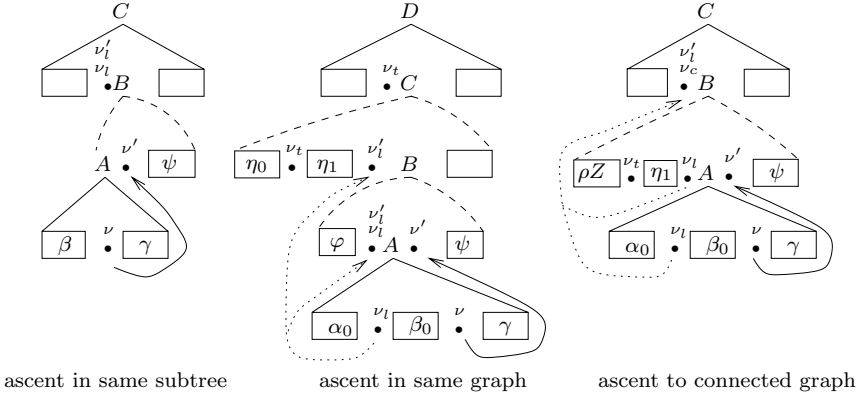


Fig. 5. Illustration of single context-recovery steps

or, if performed and subsequently resumed, no truncation (to h pairs) has taken place, then all (ν'_l, ν') such that $\nu'_l \xrightarrow{\alpha} \nu_l$ and $\nu'_l \xrightarrow{A} \nu'$ are exact. However, in case of truncation some of such (ν'_l, ν') may be in excess, resulting in precision loss.

- b) Otherwise current subgraph is connected⁴, i.e., $\kappa = \kappa_1(\nu_c, \nu_t)$. There are two possibilities:
 - i. ν' is lower than ν_t (middle tree in Fig. 5). According to the discussion in Sect. 2.5, only those ν'_l on a downward ε -skip path η from ν_t are precise.
 - ii. Otherwise ν' can only be at the same level as ν_t (rightmost tree in Fig. 5). The connected subgraph is resumed, i.e., last pair is removed from current sequence and ν_t is replaced by ν' .

The following function performs such single context-recovery steps:

$$\hat{\theta}(\kappa(\nu_l, \nu)) = \begin{cases} \{ \kappa_1(\nu_c, \nu') \mid \nu_t \xrightarrow{\eta} \nu'_l, \nu' = [B \rightarrow \rho Z \eta A \cdot \psi]_q, \eta \Rightarrow^* \varepsilon, (\nu_l, \nu) \uparrow \uparrow (\nu'_l, \nu') \} \\ \cup \{ \kappa(\nu'_l, \nu') \mid \nu_t \xrightarrow{\eta \varphi} \nu'_l, \nu' = [B \rightarrow \varphi A \cdot \psi]_q, \eta \varphi \Rightarrow^* \varepsilon, (\nu_l, \nu) \uparrow \uparrow (\nu'_l, \nu') \} & \text{if } \kappa = \kappa_1(\nu_c, \nu_t) \\ \{ (\nu'_l, \nu') \mid (\nu_l, \nu) \uparrow \uparrow (\nu'_l, \nu') \} & \text{otherwise,} \end{cases}$$

such that $(\nu_l, \nu) \uparrow \uparrow (\nu'_l, \nu')$ iff $\nu'_l \xrightarrow{\alpha} \nu_l \xrightarrow{\beta} \nu = [A \rightarrow \alpha \beta \cdot \gamma]_q$, $\nu'_l \xrightarrow{A} \nu'$, and $\gamma \Rightarrow^* \varepsilon$.

Its closure, which allows any number of context recovering steps, is defined as the minimal set such that $\hat{\theta}^*(\kappa) = \{ \kappa \} \cup \{ \kappa'' \mid \kappa'' \in \hat{\theta}(\kappa'), \kappa' \in \hat{\theta}^*(\kappa) \}$.

3 Construction of Lookahead Automata

States r in the lookahead DFA correspond to sets J_r of *lookahead items*. A lookahead item $\mu = [j, \kappa]$ will describe a current sequence (i.e., current position

⁴ In this case, $\alpha \Rightarrow^* \varepsilon$, since we always have $\nu_t \xrightarrow{\eta} \nu_l, \eta \Rightarrow^* \varepsilon$.

and its recovery context) κ for some action j in conflict. By convention, $j = 0$ for a shift action, and $j = i$ for a reduction according to $A \xrightarrow{i} \alpha$.

3.1 Transitions from Lookahead States

The following function computes the next set of lookahead items after some terminal-symbol transition:

$$\begin{aligned} \Theta_h(J_r, a) = & \\ & \hat{\Theta}(\{[j, \kappa(\nu_l, \nu')] \mid [j, \kappa(\nu_l, \nu)] \in J_r, \nu \xrightarrow{\beta Y} \nu' = [A \rightarrow \alpha X \beta Y \cdot \gamma]_q, \beta \Rightarrow^* \varepsilon, Y \Rightarrow^* a\} \\ & \cup \{[j, \kappa(\nu_c, \nu_t)(\nu_l, \nu) : h] \mid [j, \kappa(\nu_c, \nu_t)] \in J_r, \nu_t \xrightarrow{\eta} \nu_l \xrightarrow{\beta Y} \nu = [A \rightarrow \beta Y \cdot \gamma]_q, \\ & \eta \beta \Rightarrow^* \varepsilon, Y \Rightarrow^* a, \gamma \Rightarrow^* bx\}). \end{aligned}$$

The first subset corresponds to the case in which we are simply moving along the right-hand side of some rule, or we descend in a subtree to immediately return to the current level. To avoid a graph connection in the latter situation is of practical interest⁵.

The second subset corresponds to a true descent in the derivation tree, i.e., exploration will continue in it. Thus, a graph connection is performed, in order to correctly resume after the subtree exploration.

Finally, function $\hat{\Theta}$ performs all necessary ε -skip upward context-recovery including removal of useless items⁶:

$$\hat{\Theta}(J_r) = \{[j, \kappa'(\nu_l, \nu)] \mid \kappa'(\nu_l, \nu) \in \hat{\theta}^*(\kappa), [j, \kappa] \in J_r, \nu = [A \rightarrow \alpha \cdot \beta]_q, \beta \Rightarrow^* ax\}.$$

3.2 Initial LR(0)-Conflict Lookahead Item Sets

Let r_0^q be the initial lookahead state associated with some conflict-state q . Its associated set of lookahead items can be computed as follows.

$$J_{r_0^q} = \{[0, (\nu, \nu)] \mid K_q \ni \nu \xrightarrow{a} \nu'\} \cup \hat{\Theta}(\{[i, (\nu_l, \nu')]\mid A \xrightarrow{i} \alpha, \nu_l \xrightarrow{A} \nu', \nu_l \xrightarrow{\alpha} \nu \in K_q\}).$$

As for the definition of the transition function, an upward ε -skip is applied.

3.3 Inadequacy Condition

A grammar \mathcal{G} is inadequate iff, for some lookahead state r ,

$$\exists [j, \kappa], [j', \kappa] \in J_r, j \neq j'.$$

⁵ In order to make grammars more readable, productions such as *type-name* \rightarrow IDENT and *var-name* \rightarrow IDENT are frequently used. Rejecting such grammars because the graph connection looses precise context would be unfortunate.

⁶ They are now useless for subsequent transitions if $\text{First}(\beta) = \{\varepsilon\}$.

Since two lookahead items with the same κ will follow exactly the same continuations and thus accept the same language, we can predict that discrimination amongst their actions will be impossible. Such grammars are rejected. Otherwise, a correct parser is produced.

Since those κ exactly code up to at least the first h terminals of right-hand context, inadequacy condition implies that the grammar is not LALR(h).

3.4 Construction Algorithm

In each state r of the lookahead DFA, action j can be decided on symbol a if all lookahead items $[j_i, \kappa_i(\nu_i, [A_i \rightarrow \alpha_i \cdot \beta_i]_{q_i})]$ in J_r with $\text{First}(\beta_i) \ni a$ share the same action j . Otherwise, a transition is taken on a to continue lookahead exploration. Accordingly, the following construction algorithm computes the lookahead DFA's action-transition table At . Obviously, only new item sets (corresponding to new states) need to be incorporated to *Set-list* for subsequent processing.

DFA(h) GENERATOR:

```

for each conflict-state  $q$  do
  initialize Set-list with  $J_{r_0^q}$ 
  repeat
    let  $J_r$  be next item-set from Set-list
     $Ps := \{(j, a) \mid [j, \kappa(\nu_l, [A \rightarrow \alpha \cdot \beta]_q)] \in J_r, a \in \text{First}(\beta)\}$ 
    for  $a \in T \mid \exists (j, a) \in Ps$  do
      if  $\nexists (j', a) \in Ps \mid j' \neq j$  then  $At(r, a) := j$ 
      else  $J_{r'} := \Theta_h(J_r, a)$ ; add  $J_{r'}$  to Set-list;  $At(r, a) := \text{goto } r'$ 
  until end of Set-list or inadequacy (rejection)

```

4 Example

We will illustrate our method with a simplified grammar for HTML forms, as the one discussed in [13]. Forms are sequences of tagged values:

```

Company=BigCo
address=1000 Washington Ave
NYC NY 94123

```

Since a value can span several lines, the end of line is not only a delimiter between tag-value pairs. And since letters can belong to a value, a sequence of letters represents a tag only when immediately followed by an = sign. Thus, in order to know if a delimiter ends a tag-value pair, or if it belongs to the value being processed, an unbounded number of characters must be read. We shall use the following example grammar \mathcal{G}_f for HTML forms, in which l stands for any letter, and c stands for any character other than a letter or the = sign (separately including space and end-of-line does not basically modify the problem but complicates the example).

$$\begin{array}{llll}
 S' \xrightarrow{1} S \mid & S \xrightarrow{2} F & S \xrightarrow{3} S c F & F \xrightarrow{4} T = V \\
 T \xrightarrow{5} l & T \xrightarrow{6} T l & V \xrightarrow{7} \varepsilon & V \xrightarrow{8} V l \quad V \xrightarrow{9} V c
 \end{array}$$

4.1 Solutions with Commonly Used Parser Generators

It is easy to see that G_f is neither $\text{LR}(k)$ nor $\text{LL}(k)$ for any k [7].

The ANTLR solution, as given in [13], can be roughly described as follows (terminals are in capitalized letters):

```
// parser
form      : ( TAG string )+          ;
string    : ( CHAR )+              ;
// scanner
TAG       : ( 'a' .. 'z' | 'A' .. 'Z' )+ ;
FORMTOKEN : ( TAG '=' ) ==> TAG '='  {$setType(TAG);} // predicate
          | .                      {$setType(CHAR);} ;
```

This solution is rather complex and tricky: it needs to define a scanner that uses a syntactic predicate (and thus reads characters until an = or a non-letter is found) to set the token kind (TAG, whose value is a letter sequence, or CHAR, whose value is a single character). In the proper “grammar” (defined by the parser), no delimiter ends a tag-value pair, what makes the grammar ambiguous, e.g., $t=xyz=v$ can be interpreted in two possible ways, as $t=x \ yz=v$ or as $t=xy \ z=v$, since tags are defined as letter sequences followed by =. Parsing is unambiguous only because the scanner reads the whole tag when encountering its first letter.

Furthermore, this solution is inefficient: for each letter of a letter sequence in a value, the predicate is checked, and, in the worst case, parsing time is $O(n^2)$ on the input length.

No better solution can be easily found with *Yacc*; TAG and CHAR are tokens, and the trick is to let *Lex* do the work in a way analogous to ANTLR:

```
[a-zA-Z]+/= return TAG; /* TAG only if followed by an = */
.          return CHAR;
```

Again, scanning can be $O(n^2)$. Means offered by *Yacc* to resolve conflicts (token priority) are of no help. Letters can be read only once by adding

```
[a-zA-Z]+/[!=] return CHAR; /* CHAR only if not followed by an = */
```

to the *Lex* code. This simply shows that making letter sequences a token is not enough, if done independently of the right-hand context.

4.2 The Bounded Graph-Connect LRR Solution

G_f is LR-regular because there exist discriminant right-hand regular languages that allow to decide if next character is a form separator, or if it belongs to the value being processed. These languages can be respectively described by the regular expressions $\neg|cl^+=$ and $l|cl^*(c|\neg)$. A user of ANTLR (or *Yacc+Lex*) should give these regular expressions as syntactic predicates for a clean solution.

⁷ One can argue that, on this example, sequences of letters can be a token. This simply would result in an LALR(3) or LL(3) grammar, that neither *Yacc* nor ANTLR can handle without help from the user, as shown here.

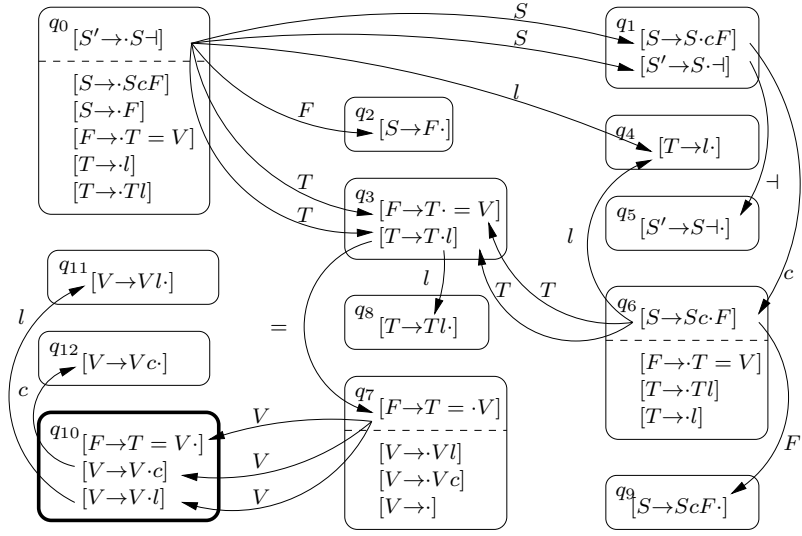


Fig. 6. LR(0) states and kernel-item graph for grammar \mathcal{G}_f

Finding these regular expressions is not that easy, especially for someone not trained in compiler/translator writing.

Let us see how our method correctly solves the problem, without user help. Figure 6 shows the corresponding LR(0) states and their underlying graph of kernel items. This automaton has one conflict state, q_{10} . Its (kernel) items are used for the initial state of a lookahead DFA, whose construction (for $h = 1$) is summarized in Fig. 7. Here is how $\hat{\theta}^*$ works for $(\nu, \nu), \nu = [F \rightarrow T = V \cdot]_{q_{10}}$ in r_0 :

- On one hand, $[S' \rightarrow \cdot S \dashv]_{q_0} \xrightarrow{T=V} \nu$, $[S' \rightarrow \cdot S \dashv]_{q_0} \xrightarrow{F} [S \rightarrow F \cdot]_{q_2}$, followed by a new context-recovery step: $[S' \rightarrow \cdot S \dashv]_{q_0} \xrightarrow{S} [S' \rightarrow S \cdot \dashv]_{q_1}$ and $[S' \rightarrow \cdot S \dashv]_{q_0} \xrightarrow{ScF} [S \rightarrow ScF \cdot]_{q_9}$.
- On the other hand, $[S \rightarrow Sc \cdot F]_{q_6} \xrightarrow{T=V} \nu$, and $[S \rightarrow Sc \cdot F]_{q_6} \xrightarrow{F} [S \rightarrow ScF \cdot]_{q_9}$; again, another context-recovery step takes place and, since $[S' \rightarrow \cdot S \dashv]_{q_0} \xrightarrow{ScF} [S \rightarrow ScF \cdot]_{q_9}$, we have the same transitions on S as above.

According to current-position nodes in the resulting lookahead item set J_{r_0} , lookahead symbol \dashv is only associated to reduction 4, and l to action 0 (shift), so they can be decided. On the other hand, since c does not allow to decide, a transition is needed to r_1 .

Decisions and transitions are likewise computed for the remaining states, resulting in the automaton shown in Fig. 8. In r_1 a transition on l to r_2 is needed because $l \in \text{First}(F)$, while for r_2 some graph transitions are actually

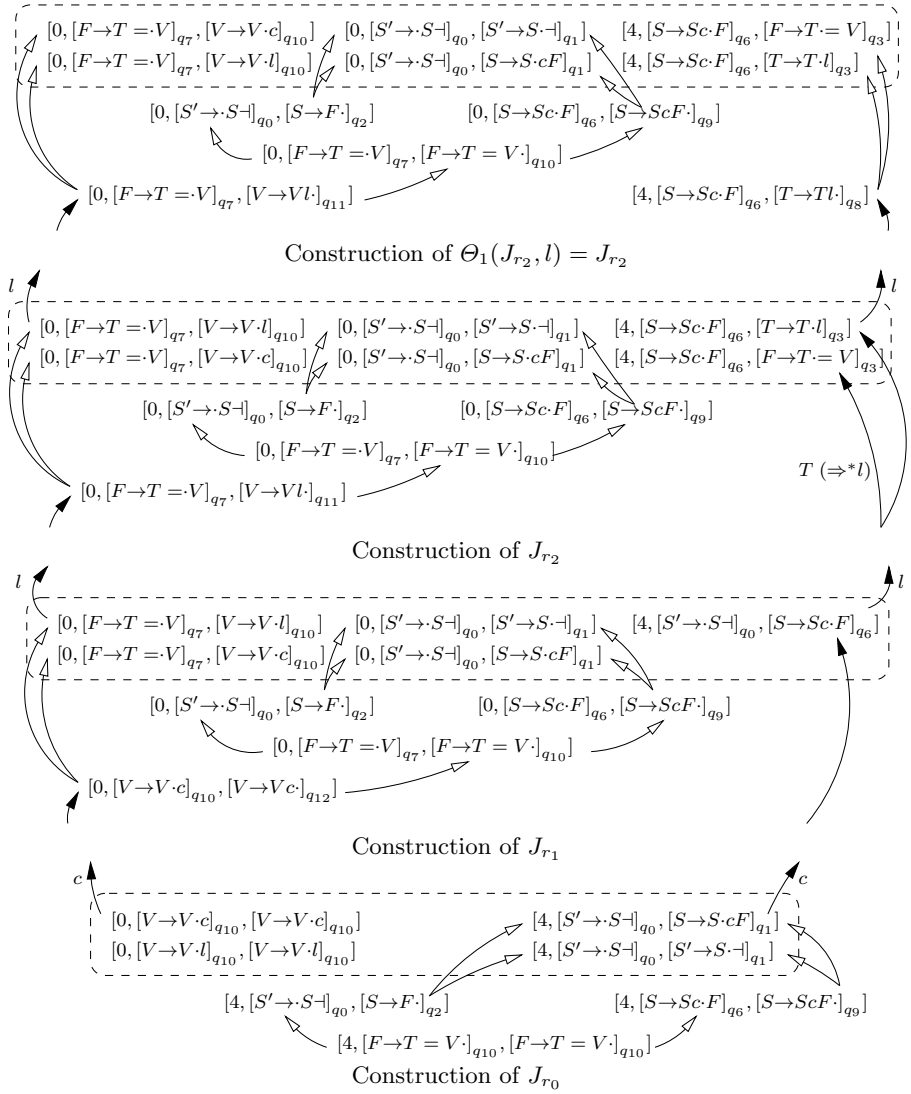


Fig. 7. DFA construction for grammar \mathcal{G}_f

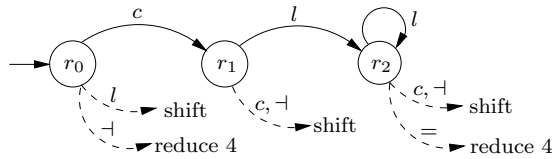


Fig. 8. Lookahead DFA for G_f

taken on T , since $T \Rightarrow^* l$. In this latter state, these downward transitions imply a connect followed by truncation to $h = 1$, although the precision loss will not affect the ability to resolve the conflict. Finally, $\Theta_1(J_{r_2}, l) = J_{r_2}$, what produces a loop in the lookahead DFA, and terminates the construction.

The reader might like to verify that time complexity for parsing is linear on input length. Following the automaton, it is easy to verify that each input character is checked at most two times. For a sequence $c_0 l_1 \cdots l_n c_1 \cdots c_m l_{n+1} \cdots$, $c_1 \neq =$, $c_0 l_1 \cdots l_n c_1$ is checked a first time, and c_0 is shifted. Next, the l_i are checked, and then shifted, one at a time. Finally, $c_i c_{i+1}$, $i = 1, \dots, m - 1$ are checked and c_i is shifted, until input is $c_m l_{n+1} \cdots$, which brings back to the first step.

Note that GLR, while producing a nondeterministic parser, gives the same complexity: it maintains two stacks as long as it cannot decide whether a letter belongs to a value or to a tag.

5 Conclusions

Available parser generation tools are not sufficiently powerful to automatically produce parsers for computer language grammars, while previous attempts to develop LR-regular parsers have failed to guarantee sufficient parsing power, since their context recovery is based on LR(0) states only.

Recent research on NDR parsing has provided with an item graph-connecting technique that can be applied to the construction of lookahead DFA to resolve LR conflicts. The paper shows how to build such DFA from an LR(0) automaton by using the dependency graph amongst kernel-items. By combining ε -skip with compact graph-connect coding, we overcome the difficulties of previous approaches. As a result, largely improved context-recovery is obtained while staying relatively simple.

While less powerful than NDR, these LRR parsers naturally preserve the properties of canonical derivation, correct prefix, and linearity. They accept a wide superset of LALR(h) grammars, including grammars not in LR, and thus permit to build practical, automatic generation tools for efficient and reliable parsers.

References

1. J. Aycock and R. N. Horspool. Faster generalized LR parsing. In S. Jähnichen, editor, *Compiler Construction. 8th International Conference, CC'99*, Lecture Notes in Computer Science #1575, pages 32–46. Springer, 1999.
2. T. P. Baker. Extending look-ahead for LR parsers. *J. Comput. Syst. Sci.*, 22(2):243–259, 1981.
3. M. E. Bermudez and K. M. Schimpf. Practical arbitrary lookahead LR parsing. *Journal of Computer and System Sciences*, 41:230–250, 1990.
4. P. Boullier. *Contribution à la construction automatique d'analyseurs lexicographiques et syntaxiques*. PhD thesis, Université d'Orléans, France, 1984. In French.
5. K. Čulik II and R. Cohen. LR-regular grammars — an extension of LR(k) grammars. *J. Comput. Syst. Sci.*, 7:66–96, 1973.
6. J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, Feb. 1970.
7. J. Farré and J. Fortes Gálvez. A basis for looping extensions to discriminating-reverse parsing. In S. Yu, editor, *Fifth International Conference on Implementation and Application of Automata, CIAA 2000*. To appear in LNCS. Springer.
8. J. Gosling, B. Joy, and G. Steele. *The JavaTM Language Specification*. Addison-Wesley, 1996.
9. D. Grune and C. J. H. Jacobs. A programmer-friendly LL(1) parser generator. *Software—Practice and Experience*, 18(1):29–38, Jan. 1988.
10. S. Heilbrunner. A parsing automata approach to LR theory. *Theoretical Computer Science*, 15:117–157, 1981.
11. A. Johnstone and E. Scott. Generalised recursive descent parsing and follow-determinism. In K. Koskimies, editor, *Compiler Construction. 7th International Conference, CC'98*, Lecture Notes in Computer Science #1383, pages 16–30. Springer, 1998.
12. B. Lang. Deterministic techniques for efficient non-deterministic parsers. In J. Loeckx, editor, *Automata, Languages and Programming*, Lecture Notes in Computer Science #14, pages 255–269. Springer, 1974.
13. T. J. Parr. We are talking really big lexical lookahead here. [www.antlr.org/articles.html](http://wwwantlr.org/articles.html).
14. T. J. Parr and R. W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software—Practice and Experience*, 25(7):789–810, July 1995.
15. B. Seité. A Yacc extension for LRR grammar parsing. *Theoretical Computer Science*, 52:91–143, 1987.
16. S. Sippu and E. Soisalon-Soininen. *Parsing Theory*. Springer, 1988–1990.
17. T. G. Szymanski and J. H. Williams. Non-canonical extensions of bottom-up parsing techniques. *SIAM J. Computing*, 5(2):231–250, June 1976.
18. M. Tomita. *Efficient Parsing for Natural Language*. Kluwer, 1986.
19. T. A. Wagner and S. L. Graham. Incremental analysis of real programming languages. *ACM SIGPLAN Notices*, 32(5):31–43, 1997.

Array Unification: A Locality Optimization Technique

Mahmut Taylan Kandemir

Computer Science and Engineering Department
The Pennsylvania State University
University Park, PA 16802-6106, USA
E-mail: kandemir@cse.psu.edu
WWW: <http://www.cse.psu.edu/~kandemir>

Abstract. One of the key challenges facing computer architects and compiler writers is the increasing discrepancy between processor cycle times and main memory access times. To alleviate this problem for a class of array-dominated codes, compilers may employ either control-centric transformations that change data access patterns of nested loops or data-centric transformations that modify the memory layouts of multi-dimensional arrays. Most of the layout optimizations proposed so far either modify the layout of each array independently or are based on explicit data reorganizations at runtime.

This paper describes a compiler technique, called *array unification*, that automatically maps multiple arrays into a single data (array) space to improve data locality. We present a mathematical framework that enables us to systematically derive suitable mappings for a given program. The framework divides the arrays accessed by the program into several groups and each group is transformed to improve spatial locality and reduce the number of conflict misses. As compared to the previous approaches, the proposed technique works on a larger scope and makes use of independent layout transformations as well whenever necessary. Preliminary results on two benchmark codes show significant improvements in cache miss rates and execution time.

1 Introduction

Processor cycle time continues to decrease at a much faster rate than main memory access times, making the cache memory hierarchy performance critical. To address this issue, conventional cache management techniques must be supported by software-oriented approaches. Throughout the years, several compiler techniques have been proposed and implemented with the objective of making memory access patterns of applications more cache-friendly. These include modifications to control structures (e.g., nested loops), careful placement of load/store operations, and reformatting memory layouts for scalar, multi-field (e.g., records), and array variables.

Modifications to control structures and smart load/store placement techniques are known to be bounded by inherent data dependences in the code.

Reformatting memory layouts, on the other hand, is less problematic as it does not change the data dependence structure of the computation. In particular, recent years have witnessed a vast group of layout optimization techniques that target at codes with inherent data locality problems such as regular numerical codes that manipulate large multi-dimensional arrays [11,9] or codes that utilize pointer structures (e.g., linked lists and trees) [1].

Previous research on data transformations for numerical codes focussed primarily on transforming a single array at a time. This special case is interesting mainly because it may enable improved spatial locality for each transformed array. For instance, the studies presented by Leung and Zahorjan [11], Cierniak and Li [2], and Kandemir et al. [9] fall into this category. Most of these transformations, like many loop-oriented optimization techniques, target only the reduction of capacity misses, namely the misses that are due to small cache sizes. However, in particular, in caches with low associativity, conflict misses can present a major obstacle to realizing good cache locality even for the codes that have specifically been optimized for data locality, thereby precluding effective cache utilization. On top of this, as opposed to capacity misses, the degree of conflict misses can vary greatly with slight variations in array base addresses, problem (input) sizes, and cache line (block) sizes [17].

In this paper, we discuss a data space transformation technique, which we call *array unification*, that transforms a number of array variables simultaneously. It achieves this by mapping a set of arrays into a common array space and replacing all references to the arrays in this set by new references to the new array space. One of the objectives of this optimization is to eliminate inter-variable conflict misses, that is, the conflict misses that are due to different array variables. Specifically, we make the following contributions:

- We present a framework that enables an optimizing compiler to map multiple array variables into a single data space. This is the *mechanism aspect* of our approach and leverages off the work done in previous research on data (memory layout) transformations.
- We present a global strategy that decides which arrays in a given code should be mapped into common data space. This is the *policy aspect* of our approach and is based on determining the temporal relations between array variables.
- We present experimental results showing that the proposed approach is successful in reducing the number of misses and give experimental data showing the execution time benefits.
- We compare the proposed approach to previous locality optimizations both from control-centric domain [19] and data-centric domain [11,9].

The rest of this paper is organized as follows. Section 2 gives the background on representation of nested loops, array variables, and data transformations and sets the scope for our research. Section 3 presents the array unification in detail. Section 4 introduces our experimental platform and the benchmark codes tested, and presents performance numbers. Section 5 discusses related work and Section 6 summarizes the contributions and gives an outline of future work.

2 Background

Our focus is on affine programs that are composed of scalar and array assignments. Data structures in the program are restricted to be multidimensional arrays and control structures are limited to sequencing and nested loops. Loop nest bounds and array subscript functions are affine functions of enclosing loop indices and constant parameters. Each iteration of the nested loop is represented by an *iteration vector*, \mathbf{I}' , which contains the values of the loop indices from outermost position to innermost. Each array reference to an m -dimensional array U in a nested loop that contains n loops (i.e., an n -level nested loop) is represented by $R_u \mathbf{I} + \mathbf{o}_u$, where \mathbf{I} is a vector that contains loop indices. For a specific $\mathbf{I} = \mathbf{I}'$, the data element $R_u \mathbf{I}' + \mathbf{o}_u$ is accessed. In the remainder of this paper, we will write such a reference as a pair $\{R_u, \mathbf{o}_u\}$. The $m \times n$ matrix R_u is called the *access (reference) matrix* [19] and the m -dimensional vector \mathbf{o}_u is called the *offset vector*. Two references (possibly to different arrays) $\{R_u, \mathbf{o}_u\}$ and $\{R_{u'}, \mathbf{o}_{u'}\}$ are called *uniformly generated references* (UGR) [6] if $R_u = R_{u'}$.

Temporal reuse is said to occur when a reference in a loop nest accesses the same data in different iterations. Similarly, if a reference accesses nearby data, i.e., data residing in the same coherence unit (e.g., a cache line or a memory page), in different iterations, we say that *spatial reuse* occurs.

A data transformation can be defined as a transformation of array index space [11]. While, in principle, such a transformation can be quite general, in this work, we focus on the transformations that can be represented using linear transformation matrices. If \mathcal{G}_u is the *array index space* (that is, the space that contains all possible array indices within array bounds) for a given m -dimensional array U , a data transformation causes an element $\mathbf{g} \in \mathcal{G}$ to be mapped to $\mathbf{g}' \in \mathcal{G}'$, where \mathcal{G}' is the new (transformed) array index space. Such a data transformation can be represented by a pair $\{M_u, \mathbf{f}_u\}$, where M_u is an $m \times m'$ matrix and \mathbf{f}_u is an m' -dimensional vector. The impact of such a data transformation is that a reference such as $\{R_u, \mathbf{o}_u\}$ is transformed to $\{R'_u, \mathbf{o}'_u\}$, where $R'_u = M_u R_u$ and $\mathbf{o}'_u = M_u \mathbf{o}_u + \mathbf{f}_u$. Most of the previous data transformation frameworks proposed in literature handle the special case where $m = m'$. The theory of determining suitable data transformation for a given array variable and post-transformation code generation techniques have been discussed in [11][8].

3 Array Unification

3.1 Approach

Consider the following loop nest that accesses two one-dimensional arrays using the same subscript function:

```
for  $i = 1, N$ 
   $b+ = U_1[i] + U_2[i]$ 
```

If considered individually, each of the references in this nest has perfect spatial locality as successive iterations of the i loop access consecutive elements from each array. However, if, for example, the base addresses of these arrays happen to cause a conflict in cache, the performance of this nest can degrade dramatically. In fact, in this code, it might even be possible to obtain a miss rate of 100%. The characteristic that leads to this degradation is that, between two successive accesses to array U_1 , there is an intervening access from U_2 , and vice versa. In other words, there is an iteration distance of 1 between successive uses of the same cache line, and during this duration, the said cache line *may* be evicted from cache due to the intervening access to the other array.

Let us now consider the following mapping of arrays U_1 and U_2 to the common data space (array) X .

$$U_1[i] \longrightarrow X[2i - 1] \text{ and } U_2[i] \longrightarrow X[2i],$$

in which case we can re-write the loop nest as follows:¹

```
for  $i = 1, N$ 
   $b+ = X[2i - 1] + X[2i]$ 
```

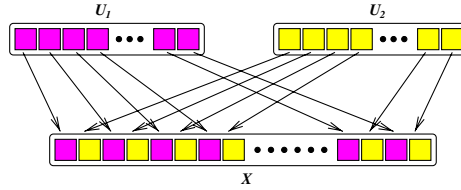


Fig. 1. Unification of two one-dimensional arrays.

A pictorial representation of these mappings is given in Figure 1. Note that in this new nest, for a given loop iteration, two references access two consecutive array elements; that is, the reuse distance between two successive accesses to the same cache line is potentially reduced to 0. Note that the same transformation can be done with multidimensional arrays as well. As an example, consider the following two-level nested loop:

```
for  $i = 1, N$ 
  for  $j = 1, N$ 
     $c+ = U_1[i][j] + U_2[i][j]$ 
```

Again, considering each reference separately, we have (potentially) perfect spatial locality (assuming that the arrays have a *row-major* memory layout).

¹ Note that, here, for the first transformation, we have $M_{u_1} = [2]$ and $f_{u_1} = -1$ whereas for the second transformation, $M_{u_2} = [2]$ and $f_{u_2} = 0$.

However, conflict misses may prevent this nest from attaining this locality at runtime. This would occur, for example, when two references have the same linearized stride and have the base addresses that map close together in cache. If we use the transformations

$$U_1[i][j] \longrightarrow X[i][2j - 1] \text{ and } U_2[i][j] \longrightarrow X[i][2j],$$

we obtain the following nest:

```

for i = 1, N
  for j = 1, N
    c+ = X[i][2j - 1] + X[i][2j];
    
```

Note that this transformation can be viewed as converting two $N \times N$ arrays to a single $N \times 2N$ array. In general, if we want to transform k two-dimensional arrays $U_i[N][N]$ ($1 \leq i \leq k$)—all accessed with the same subscript expression $[i][j]$ —to a single two-dimensional array $X[N][kN]$, we can use the following generic data transformation:

$$M_{u_i} = \begin{bmatrix} 1 & 0 \\ 0 & k \end{bmatrix}, \quad \mathbf{f}_{u_i} = \begin{bmatrix} 0 \\ i - k \end{bmatrix}.$$

Note that this transformation strategy can also be used when references have subscript expressions of the form $[i \mp a][j \mp b]$. So far, we have implicitly assumed that the references accessed in the nest are uniformly generated. If two arrays U_1 and U_2 are accessed using references $\{R_{u_1}, \mathbf{o}_{u_1}\}$ and $\{R_{u_2}, \mathbf{o}_{u_2}\}$, respectively, where R_{u_1} and R_{u_2} are *not* necessarily the same, we need to select two data transformations $\{M_{u_1}, \mathbf{f}_{u_1}\}$ and $\{M_{u_2}, \mathbf{f}_{u_2}\}$ such that the transformed references $\{M_{u_1}R_{u_1}, M_{u_1}\mathbf{o}_{u_1} + \mathbf{f}_{u_1}\}$ and $\{M_{u_2}R_{u_2}, M_{u_2}\mathbf{o}_{u_2} + \mathbf{f}_{u_2}\}$ will access consecutive data items for a given iteration. Also, when considered individually, each reference should have good spatial locality. As an example, for the following nest, we can use the transformations

$$M_{u_1} = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}, \quad \mathbf{f}_{u_1} = \begin{bmatrix} 0 \\ -1 \end{bmatrix};$$

$$M_{u_2} = \begin{bmatrix} 0 & 1 \\ 2 & 0 \end{bmatrix}, \quad \mathbf{f}_{u_2} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

```

for i = 1, N
  for j = 1, N
    c+ = U_1[i][j] + U_2[j][i]
    
```

After these transformations, the references inside the nest will be $X[i][2j - 1]$ and $X[i][2j]$ instead of $U_1[i][j]$ and $U_2[j][i]$, respectively, exhibiting high group-spatial reuse. *It should be noted that such transformations actually involve both independent array transformations and array unification.* To see this, we can think of M_{u_2} as a composition of two different transformations. More specifically,

$$M_{u_2} = \begin{bmatrix} 0 & 1 \\ 2 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

It should be noted that array unification can also be used to optimize locality performance of references with non-affine subscript functions. In fact, as noted by Leung and Zahorjan [11], data layout optimizations promise a wider applicability than loop-oriented techniques as far as non-affine references are concerned. This is because, unlike loop restructuring, the legality of data transformations do not rely on dependence analysis whereas the existence of even a single non-affine reference may prevent the candidate loop transformation which would otherwise improve data locality.

Let us consider the loop shown below assuming that $f(\cdot)$ is a non-affine expression (which might even be an index array).

```
for i = 1, N
  b+ = U1[f(i)] + U2[f(i)]
```

If considered individually, each reference here may have very poor locality as there is no guarantee that the function $f(\cdot)$ will take successive values for successive iterations of i . In the worst case, this loop can experience $2N$ cache misses. Now, consider the following data transformations:

$$U_1[f(i)] \longrightarrow X[f(i), 0] \text{ and } U_2[f(i)] \longrightarrow X[f(i), 1],$$

assuming that the new array X has a row-major layout. It is easy to see that, due to the high probability that $X[f(i), 1]$ and $X[f(i), 2]$ will use the same cache line, we might be able to reduce the number of misses to N . Notice that the data transformations used here can be represented by

$$M_{u_1} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \mathbf{f}_{u_1} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \text{ and } M_{u_2} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \mathbf{f}_{u_2} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

It should be noted that this transformation can reduce capacity as well as conflict misses.

There is an important difference between such transformations and those considered earlier for the affine-references case. The data transformation matrices used for non-affine references need to be non-square. Of course, nothing prevents us to use non-square transformations for affine references as well. Consider the two-level nested loop shown above that has the references $U_1[i][j]$ and $U_2[i][j]$. Instead of using the transformations

$$U_1[i][j] \longrightarrow X[i][2j - 1] \text{ and } U_2[i][j] \longrightarrow X[i][2j],$$

we could have easily used

$$U_1[i][j] \longrightarrow X[i][j][0] \text{ and } U_2[i][j] \longrightarrow X[i][j][1].$$

Therefore, in general, we have the options of using a two-dimensional array of size $N \times 2N$ or a three-dimensional array of size $N \times N \times 2$. The theory developed in the rest of the paper can easily be adapted to the latter case as well.

A group of arrays with a unification characteristic is called an *interest group*. The *unification characteristic* of the arrays in the same interest group is that

they have the same dimensionality and each dimension has the same extent and the arrays are accessed with the *same frequency* in the innermost loop. The first two requirements are obvious. As for the third requirement, we say that two references are accessed with the same frequency if (in their subscript functions) they use exactly the same set of loop index variables. For instance, for a three-level nested loop with indices i , j , and k from top, the references $U_1[i+k][j]$ and $U_2[i][j-k]$ are accessed with the same frequency whereas the references $U_1[i][j]$ and $U_2[j][k]$ are *not* accessed with the same frequency. The problem with this last pair of references is that the reference $U_2[j][k]$ traverses the array at a much faster rate than the reference $U_1[i][j]$ (as the former has the innermost loop index k). It is known that a majority of the severe inter-variable conflict misses occur between the references that are accessed with the same frequency [17]. Later in the paper, we discuss more relaxed unification characteristics. Let us assume for now, without loss of generality, that we have only one interest group (the approach to be presented can be applied to each interest group independently), and, that each nested loop in a given code accesses a subset of the arrays in this interest group. Our objective is to determine a unification strategy so that the overall data locality of the code is improved. Our approach makes use of a graph structure where *nodes* represent *array variables* and *edges* represent the *transitions* between them. Let us first consider the example code fragment below to motivate our approach.

```

for i = 1, N
  b+ = U1[i] + U2[i]
for i = 1, N
  c+ = U3[i] + c

```

If we transform only the arrays U_1 and U_2 using the mappings

$$U_1[i] \longrightarrow X[2i-1] \text{ and } U_2[i] \longrightarrow X[2i],$$

we will have the statement $b+ = X[2i-1] + X[2i]$ in the first nest. It is easy to see that, after these transformations, we have perfect spatial locality in both the nests. On the other hand, if we transform only U_1 and U_3 using the mappings

$$U_1[i] \longrightarrow X[2i-1] \text{ and } U_3[i] \longrightarrow X[2i],$$

we will have $b+ = X[2i-1] + U_2[i]$ in the first nest and $c+ = X[2i] + c$ in the second. The problem with this transformed code is that the references $X[2i-1]$ and $X[2i]$ iterate over the array using a step size of 2 and they are far apart from each other. It is very likely that a cache line brought by $X[2i-1]$ will be kicked off the cache by the reference $U_2[i]$ in the same or a close-by iteration. Finally, let us consider transforming all three arrays using

$$U_1[i] \longrightarrow X[3i-2], U_2[i] \longrightarrow X[3i-1], \text{ and } U_3[i] \longrightarrow X[3i].$$

In this case, we will have $b+ = X[3i-2] + X[3i-1]$ in the first nest and $c+ = X[3i] + c$ in the second nest. Note that in the second nest, we traverse the

array using step size of 3 which may lead to poor locality for, say, a cache line size that can hold at most two elements. The preceding discussion shows that (1) selecting the subset of arrays to be unified (from an interest group) might be important, and that (2) it might not always be a good idea to include all the arrays (even in a given interest group) in the unification process. For instance, in the current example, the best strategy is to unify only the arrays U_1 and U_2 . It is also easy to imagine that the use of the same array variable in different loop nests can make the problem even more difficult.

3.2 Representation

In order to capture the temporal relations between array variables globally, we use an undirected graph called *array transition graph* (ATG). In a given $\text{ATG}(V, E)$, V represents the set of array variables used in the program, and there is an edge $e = (v_1, v_2) \in E$ with a weight of $w(e)$ if and only if there are $w(e)$ *transitions* between the arrays represented by v_1 and v_2 . A transition between v_1 and v_2 corresponds to the case that the array variable represented by v_2 is touched immediately after the array variable represented by v_1 is touched, or vice versa. Although an ATG can be built by instrumenting the program and counting the number of transitions between different array variables, in this paper, we construct an ATG using the information (constant or symbolic) available at compile-time.

Consider the following code fragment, whose ATG is given in Figure 2(a).

```

for i = 1, N
   $U_5[i] = (U_3[i] * U_4[i]) + (U_1[i] * U_2[i])$ 
for i = 1, N
   $U_6[i] = (U_1[i] + U_3[i]) * U_5[i]$ 
for i = 1, N
   $U_4[i] = (U_5[i] - U_6[i]) * U_1[i]$ 

```

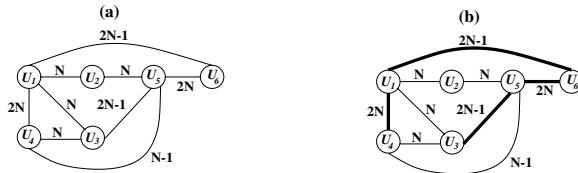


Fig. 2. (a) Address Transition Graph (ATG). (b) An important path.

As explained earlier, in a given ATG, the edge weights represent the number of transitions. For example, the weight of the edge between U_1 and U_4 is $2N$, indicating that there are $2N$ transitions between these two array variables. N of these transitions come from the first nest where U_1 is accessed immediately after

U_4 . The remaining transitions, on the other hand, come from the last nest where U_4 is written after U_1 is touched. Note that it makes no difference in which order we touch the variables as long as we touch them consecutively. An important thing to note about the ATG shown is that the edge weights differ from each other and it makes sense to unify the arrays with large edge weights as these are the arrays that are most frequently accessed one after another. For example, we have $2N$ transitions between the variables U_1 and U_4 whereas we have only $N-1$ transitions between U_4 and U_5 , which indicates that it is more important to unify U_1 and U_4 than unifying U_4 and U_5 . The problem then is to select a unification order (basically, paths in the ATG) such that as many high weight edges as possible should be covered. Note that this problem is similar to the state assignment with minimum Hamming distance problem, therefore, a polynomial-time solution is unlikely to exist. In fact, Liao shows that a similar problem (called *offset assignment* that involves selecting suitable placement order for scalar variables) is NP-complete [13]. Eisenbeis et al. [5] also use a similar graph-based representation. In the following, we present a polynomial-time heuristic which we found very effective in cases encountered in practice. Also note that in case a different memory access pattern is imposed by back-end, the code can be profiled first to extract this access pattern, and then, based on that, a suitable ATG can be constructed.

We start by observing that (after unification) a given array variable can have only *one left neighbor* and *one right neighbor*. If we take a closer look at the problem and its ATG, we can see that it is important to capture only *the paths with high weights*. After we detect a path that contains edges with high weights (henceforth called *important path*), then we can unify the array variables on this path. Note that a given ATG may lead to multiple important paths. Figure 2(b) shows an important path (using edges marked bold) that contains U_4 , U_1 , U_6 , U_5 , and U_3 . Consequently, a solution for this example is to unify these variables in that order. The transformed program is as follows:

```

for  $i = 1, N$ 
   $X[5i - 1] = (X[5i] * X[5i - 4]) + (X[5i - 3] * U_2[i])$ 
for  $i = 1, N$ 
   $X[5i - 2] = (X[5i - 3] + X[5i]) * X[5i - 1]$ 
for  $i = 1, N$ 
   $X[5i - 4] = (X[5i - 1] - X[5i - 2]) * X[5i - 3]$ 

```

3.3 Formulation

We now present our strategy for determining the important paths whose vertices are to be unified. Given an ATG and a number of important paths on it, the cost of a unification strategy can be defined as the sum of the weights of the edges that do *not* belong to any important path. Let us first make the following formal definitions:

Definition 1 *Two paths are said to be ‘disjoint’ if they do not share any vertices.*

Definition 2 A ‘disjoint path cover’ (which will be referred to as just a ‘cover’) of an $ATG(V, E)$ is a subgraph $C(V, E')$ such that, for every vertex v in C , we have $\text{degree}(v) < 3$, and there are no cycles in C ($\text{degree}(\cdot)$ denotes the number of edges incident on a vertex).

Definition 3 The ‘weight’ of a cover C is the sum of the weights of all the edges in C . The ‘cost’ of a cover C is the sum of the weights of all edges in G but not in C :

$$\text{cost}(C) = \sum_{(e \in E) \wedge (e \notin C)} w(e)$$

A cover contains a number of important paths and the array variables in each important path can be unified. It can be seen that the cost of a unification is the number of adjacent accesses to array variables that are *not* adjacent in an important path. Given the definitions above, if a maximum weight cover for an ATG is found, then that also means that the minimum cost unification has also been found. The thick lines in Figure 2(b) show a disjoint path cover for the ATG given in Figure 2(a). The cost of this cover is $5N-1$ which can be seen from the edges not in the cover.

Our unification problem can be modeled as a graph theoretic optimization problem similar to Liao’s [13] modeling of the offset assignment problem for DSP architectures and can be shown to be equivalent to the Maximum Weighted Path Cover (MWPC) problem. This problem is proven to be NP-hard. Next, a heuristic solution to the unification problem is given.

The heuristic is similar to Kruskal’s spanning tree algorithm. Taking the ATG as input, it first builds a list of sorted edges in descending order of weight. The cover to be determined initially is empty. In each iteration, an edge with the maximum weight is selected such that the selected edges never form a cycle and no node will be connected to more than two selected edges. Note that the approach iterates at most $V - 1$ times and its complexity is $O(|E| \log |E| + |V|)$.

We now focus on different unification characteristics that are more relaxed than the one adopted so far, and discuss our approach to each of them. Specifically, we will consider three cases:

- (1) Arrays that are accessed with the same frequency and have the same dimensionality and have the same dimension extents up to a permutation can be part of an important path (that is, they can be unified).
- (2) Arrays that are accessed with the same frequency and have the same dimensionality but with different dimension extents can be part of an important path.
- (3) Arrays that are accessed with the same frequency but have different dimensionalities can be part of an important path.

An example of the first case is the following two-level nested loop where arrays U_1 and U_2 are of sizes $N \times M$ and $M \times N$, respectively.


```

for  $i = 1, N$ 
  for  $j = 1, M$ 
     $b+ = U_1[i][j] + U_2[j][i]$ 

```

Note that this first case is quite common in many floating-point codes. As before, we want to transform the references $U_1[i][j]$ and $U_2[j][i]$ to $X[i][2j-1]$ and $X[i][2j-1]$, respectively. Our approach to this problem is as follows. Since the dimension extents of the arrays U_1 and U_2 are the same up to a permutation, we first use a data transformation to one of the arrays (say U_2) so that the dimension extents become the same. Then, we can proceed with our usual transformation and map both the arrays to the common domain X .

In the second case, we allow the dimension extents of the arrays to be unified to be different from each other. Consider the example below assuming that the arrays U_1 and U_2 are of $N \times N$ and $M \times M$, respectively, and $N \leq M$.

```

for  $i = 1, N$ 
  for  $j = 1, N$ 
     $b+ = U_1[i][j] + U_2[j][i]$ 

```

In this case, we can consider at least two options. First, we can unify the arrays considering the largest extent in each dimension. In the current example, this corresponds to mappings

$$U_1[i][j] \longrightarrow X[i][2j-1] \text{ and } U_2[i'][j'] \longrightarrow X[i'][2j'],$$

for $1 \leq i, j \leq N$ and $1 \leq i', j' \leq N$. On the other hand, for $N+1 \leq i', j' \leq M$, we can set $X[i'][2j']$ to arbitrary values as these elements are never accessed. A potential negative side effect of this option is that if the elements $X[i][2j-1]$ (when $N+1 \leq i, j \leq M$) happen to be accessed in a separate loop, the corresponding cache lines will be underutilized. The second option is to transform only the portions of the arrays that are used together. For instance, in our example, we can divide the second array above (U_2) into the following three regions:

$$U_{21}[i][j] (1 \leq i, j \leq N), U_{22}[i][j] (N+1 \leq i \leq M, 1 \leq j \leq N),$$

$$\text{and } U_{23}[i][j], (1 \leq i \leq N, N+1 \leq j \leq M).$$

After this division, we can unify U_1 with only U_{21} as they represent the elements that are accessed concurrently.

To handle the third case mentioned above, we can use array replication. We believe that the techniques developed in the context of cache interference minimization (e.g., [17]) can be exploited here. The details are omitted due to lack of space.

4 Experiments

The experiments described in this section were performed on a single R10K processor (195 MHz) of the SGI/Cray Origin 2000 multiprocessor machine. The

processor utilizes a two-level data cache hierarchy: there is a 32 KB primary (L1) cache and a 4 MB secondary (L2) cache (unified). The access time for the L1 cache is 2 or 3 clock cycles and that for the L2 cache is 8 to 10 clock cycles. Main memory access time, on the other hand, ranges from 300 to 1100 nanoseconds (that is 60 to 200+ cycles).

The proposed framework has been implemented within the Parafrase-2 compiler. The current implementation has the following limitations. First, it uses a restricted form of unification characteristic. It does not unify two arrays unless they are accessed with the same frequency and are of the same dimensionality and have the same extents in each dimension. Second, it works on a single procedure at a time. It does not propagate the memory layouts across procedures, instead it simply transforms the unified arrays explicitly between procedure boundaries at runtime. Finally, except for procedure boundaries, it does not perform any dynamic layout transformation. In other words, an array is never unified with different groups of arrays in different parts of the code.

We use two benchmark codes: **bmcm** and **charmm**. The **bmcm** is a *regular* array-dominated code from the Perfect Club benchmarks. We report performance numbers for six different versions of the code: **noopt**, **dopt**, **unf**, **noopt+**, **dopt+**, and **unf+**. Each version is eventually compiled using the native compiler on the Origin. **noopt** is the unoptimized (original) code, and **dopt** is the version that uses individual data transformations for each array. It is roughly equivalent to data-centric optimization schemes proposed in [8] and [2]. The version **unf**, on the other hand, corresponds to the code obtained through the technique explained in this paper. These three versions do not use the native compiler’s locality optimizations, but use back-end (low-level) optimizations. The versions **noopt+**, **dopt+**, and **unf+** are the same as **noopt**, **dopt**, and **unf**, respectively, except that the entire suite of locality optimizations of the native compiler is activated.

Figure 3 shows the execution times (seconds), MFLOPS rates, L1 hit rates, and L2 hit rates for these six versions using two different input sizes (small= ~ 6 MB and large= ~ 24 MB). We see that for both small and large input sizes, **unf** performs better than **noopt** and **dopt**, indicating that, for this benchmark, array unification is more successful than transforming memory layouts independently. In particular, with a 14 MB input size, **unf** reduced the execution time by around 67% over the **dopt** version. We also note that the technique proposed in this paper is not sufficient alone to obtain the best performance. For instance, with the large input size, **noopt+** generates 131.20 MFLOPS which is much better than 94.78 MFLOPS of **unf**. However, applying tiling brings our technique’s performance to 186.77 MFLOPS. These results demonstrate that control-centric transformations such as tiling are still important even in the existence of array unification. Finally, we also note that the **dopt** version takes very little advantage of the native compiler’s powerful loop optimizations.

We next focus on a kernel computation from an irregular application, **charmm**. This code models the molecular dynamic simulation; it simulates dynamic interactions (bonded and nonbonded) among all atoms for a specific period of time. As in the previous example, we experiment with two different input sizes

	~6 MB						~24 MB					
	noopt	noopt+	dopt	dopt+	unf	unf+	noopt	noopt+	dopt	dopt+	unf	unf+
Execution Time (sec):	2.169	0.793	1.890	0.791	1.409	0.641	49.964	7.631	32.734	6.928	10.627	5.443
MFLOPS:	56.45	176.52	63.17	176.28	85.62	215.33	20.07	131.20	30.59	145.27	94.78	186.77
L1 Hit Rate (%):	82.8	94.8	87.5	93.5	99.8	99.4	55.9	95.2	87.2	92.4	89.8	99.4
L2 Hit Rate (%):	96.9	98.7	99.6	99.3	98.6	99.6	95.2	98.0	82.0	99.3	95.8	97.9

Fig. 3. Performance results for **bmcm**.

(small= ~ 240 KB and large= ~ 12 MB). Since **dopt** version of this code is same as the **noopt** version, we omit the former from consideration. We can make two important observations from Figure 4. First, as expected, control-centric locality optimizations of the native compiler bring only a marginal benefit, most of which is due to scalar replacement. Second, the L1 miss rates for the **noopt** version are quite high due to irregular nature of the data access pattern exhibited by the kernel. A simple array unification which maps two most frequently accessed (through indexed arrays) arrays into a single array improves both miss rates and performance.

Overall, for these two codes, we find array unification to be very successful.

	~240 KB				~12 MB			
	noopt	noopt+	unf	unf+	noopt	noopt+	unf	unf+
Execution Time (sec):	0.964	0.961	0.804	0.809	240.445	240.429	188.941	187.055
MFLOPS:	127.93	128.07	146.43	145.793	119.98	119.93	167.82	170.11
L1 Hit Rate (%):	70.6	75.5	90.1	90.0	85.2	90.5	95.5	95.3
L2 Hit Rate (%):	98.1	96.0	98.3	98.0	83.1	91.8	96.3	96.8

Fig. 4. Performance results for a kernel code from **charmm**.

5 Related Work

Wolf and Lam [19] present a locality optimization framework that makes use of both unimodular loop transformations as well as tiling. Li [12] proposes a locality optimization technique that models the reuse behavior between different references to the same array accurately. McKinley, Carr, and Tseng [14] present a simpler but very effective framework based on a cost (cache miss) model. Several other researchers propose different tiling algorithms with methods to select suitable tile sizes (blocking factors) [710]. The effectiveness of these techniques is limited by the inherent data dependences in the code.

More recently, a number of researchers have addressed limitations of loop-based transformations, and proposed data layout optimizations called data transformations. Leung and Zahorjan [11] and Kandemir et al. [8] propose data transformations that apply individual layout optimizations for each array. While such transformations are known to reduce some portion of conflict misses (in addition to capacity misses) due to improved spatial locality in the inner loop positions, they are not very effective to reduce inter-variable conflict misses. To reduce the severe impact of conflict misses, Rivera and Tseng propose array padding [18].

Many research groups have also focused on combining loop and data transformations in an integrated framework. Cierniak and Li [2], Kandemir et al. [9], and O’Boyle and Knijnenburg [16] investigate the integrated use of loop and data transformations to enhance data locality. The large search space for such integrated approaches generally forces them to limit the number of potential transformations. The data-centric array unification framework used in this paper can also be integrated with loop transformations (although this issue is beyond the scope of this paper).

Recently, a number of runtime-based approaches have been proposed to improve locality of irregular array applications. Mellor-Crummey et al. [15], and Ding and Kennedy [3] present data re-ordering techniques for irregular applications. While these techniques specifically focus on irregular applications, the framework proposed in this paper can be used for both irregular and regular applications.

Ding and Kennedy [4] present an inter-array data regrouping strategy to enhance locality. The main idea is to pack useful data into cache lines so that all data elements in a cache line are consumed before the line is evicted from the cache. Our work is different from theirs in several aspects. First, their implementation does not group arrays unless they are always accessed together. In contrast, our approach has a more *global view*, and considers the entire procedure with different nests accessing different subsets of the arrays declared. Second, since we formulate the problem within a mathematical framework, we can easily integrate array unification with existing loop and data transformations. Third, it is not clear in their work how the arrays with multiple and different (nonuniformly generated) references are handled.

6 Conclusions and Future Work

We have described a data optimization scheme called array unification. Our approach is based on a mathematical framework that involves arrays, access patterns, and temporal access relations between different array variables. The general problem is formulated on a graph structure and temporal access relations are captured by this representation. Subsequently, a heuristic algorithm determines the array variables to be unified and maps them into a common array (data) space.

This work can be extended in a number of ways. First, we need a solid mechanism to integrate array unification with loop-based transformation techniques. Second, the current approach works only on a single procedure at a time. A mechanism that propagates the new array spaces (after unification) across procedure boundaries would further improve the performance.

References

1. T. Chilimbi, M. Hill, and J. Larus. Cache-conscious structure layout. In *Proc. the SIGPLAN’99 Conf. on Prog. Lang. Design and Impl.*, Atlanta, GA, May 1999.

2. M. Cierniak and W. Li. Unifying data and control transformations for distributed shared memory machines. In *Proc. SIGPLAN '95 Conf. on Programming Language Design and Implementation*, June 1995.
3. C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at runtime. In *Proc. ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation*, Georgia, May, 1999.
4. C. Ding and K. Kennedy. Inter-array data regrouping. In *Proc. the 12th Workshop on Languages and Compilers for Parallel Computing*, San Diego, CA, August 1999.
5. C. Eisenbeis, S. Lelait, and B. Marmol. The meeting graph: a new model for loop cyclic register allocation. In *Proc. the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques*, Limassol, Cyprus, June 1995.
6. D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformations. *Journal of Parallel & Distributed Computing*, 5(5):587–616, October 1988.
7. F. Irigoin and R. Triolet. Super-node partitioning. In *Proc. 15th Annual ACM Symp. Principles of Prog. Lang.*, pp. 319–329, San Diego, CA, January 1988.
8. M. Kandemir, A. Choudhary, N. Shenoy, P. Banerjee, and J. Ramanujam. A hyperplane based approach for optimizing spatial locality in loop nests. In *Proc. 1998 ACM Intl. Conf. on Supercomputing*, Melbourne, Australia, July 1998.
9. M. Kandemir, J. Ramanujam, and A. Choudhary. A compiler algorithm for optimizing locality in loop nests. In *Proc. 11th ACM Intl. Conf. on Supercomputing*, pages 269–276, Vienna, Austria, July 1997.
10. I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *Proc. SIGPLAN Conf. Programming Language Design and Implementation*, June 1997.
11. S.-T. Leung and J. Zahorjan. Optimizing data locality by array restructuring. *Technical Report TR 95-09-01*, Dept. Computer Science and Engineering, University of Washington, Sept. 1995.
12. W. Li. *Compiling for NUMA parallel machines*. Ph.D. Thesis, Cornell Uni., 1993.
13. S. Y. Liao. *Code Generation and Optimization for Embedded Digital Signal Processors*. Ph.D. Thesis, Dept. of EECS, MIT, Cambridge, Massachusetts, June 1996.
14. K. McKinley, S. Carr, and C.W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 1996.
15. J. Mellor-Crummey, D. Whalley, and K. Kennedy. Improving memory hierarchy performance for irregular applications. In *Proc. the ACM Intl. Conf. on Supercomputing*, Rhodes, Greece, June 1999.
16. M. O'Boyle and P. Knijnenburg. Integrating loop and data transformations for global optimisation. In *Intl. Conf. on Parallel Architectures and Compilation Techniques*, October 1998, Paris, France.
17. O. Temam, E. Granston, and W. Jalby. To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *Proc. the IEEE Supercomputing'93*, Portland, November 1993.
18. G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In *Proc. the 1998 ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation*, Montreal, Canada, June 1998.
19. M. Wolf and M. Lam. A data locality optimizing algorithm. In *Proc. ACM SIGPLAN 91 Conf. Prog. Lang. Design and Implementation*, pages 30–44, June 1991.

Optimal Live Range Merge for Address Register Allocation in Embedded Programs

Guilherme Ottoni¹, Sandro Rigo¹, Guido Araujo¹, Subramanian Rajagopalan²,
and Sharad Malik²

¹ University of Campinas, Campinas SP 6176, Brazil

² Princeton University, Princeton NJ 08544, USA

Abstract. The increasing demand for wireless devices running mobile applications has renewed the interest on the research of high performance low power processors that can be programmed using very compact code. One way to achieve this goal is to design specialized processors with short instruction formats and shallow pipelines. Given that it enables such architectural features, indirect addressing is the most used addressing mode in embedded programs. This paper analyzes the problem of allocating address registers to array references in loops using auto-increment addressing mode. It leverages on previous work, which is based on a heuristic that merges address register live ranges. We prove, for the first time, that the merge operation is NP-hard in general, and show the existence of an optimal linear-time algorithm, based on dynamic programming, for a special case of the problem.

1 Introduction

Complex embedded programs running on compact wireless devices has become a typical computer system nowadays. Although this trend is the result of a mature computing technology, it brings a new set of challenges. It has become increasingly hard to meet the stringent design requirements of such systems: low power consumption, high performance and small code size. Due to this design constraints, many embedded programs are written in assembly, and run on specialized embedded processors like *Digital Signal Processors* (DSPs) (e.g. ADSP 21000 [2]) or commercial CISC machines (e.g. Motorola 68000 [15]).

The increase in the size of embedded applications has put a lot of pressure towards the development of optimizing compilers for these architectures. One way to achieve this goal is to design specialized processors with short instruction formats and shallow pipelines. Since it allows such architectural improvements, indirect addressing is the most common addressing mode found in embedded programs. Again, due to cost constraints, the number of address registers available in the processor is fairly small, specially in DSPs (typically ≤ 8). This paper analyzes the problem of allocating address registers to array references in embedded program loops.

The register allocation problem has been extensively studied in the compiler literature [5,4,6,12] for general-purpose processors, but not enough work has been

done for the case of highly constrained embedded processors. *Local Reference Allocation* (LRA) is the problem of allocating address registers to array references in a basic block such that the number of address registers and instructions required to update them are minimized. LRA has been studied before in [311,13]. These are efficient graph-based solutions, when references are restricted to basic block boundaries. In particular, Leupers et al [13] is a very efficient solution to LRA. Unfortunately, not much work has been developed towards finding solutions to the global form of this problem, when array references are spread across different basic blocks, a problem we call *Global Reference Allocation* (GRA). The basic concepts required to understand GRA are described in Sect. 3. In [7] we proposed a solution to GRA, based on a technique that we call *Live Range Growth* (LRG). In Sect. 3 of this paper we give a short description of the LRG algorithm. In the current work, we extend our study of GRA in two ways. First, we perform the complexity analysis of the merge operation required by LRG, and prove it to be NP-hard (Sect. 4). Moreover, we prove the existence of an optimal linear-time algorithm for a special case of merge (Sect. 5). In Sect. 6 we evaluate the performance of this algorithm, and show that it can save update instructions. Section 7 summarizes the main results.

2 Problem Definition

Global Reference Allocation (GRA) is the problem of allocating address registers (*ar*'s) to array references such that the number of simultaneously live address registers is kept below the maximum number of such registers in the processor, and the number of new instructions required to do that is minimized. In many compilers, this is done by assigning address registers to similar references in the loop. This results in efficient code, when traditional optimizations like induction variable elimination and loop invariant removal [116] are in place. Nevertheless, assigning the same address register to different instances of the same reference does not always result in the best code. For example, consider the code fragment of Fig. 1(a). If one register is assigned to each reference, $a[i+1]$ and $a[i+2]$, two address registers are required for the loop. Now, assume that only one address register is available in the processor. If we rewrite the code from Fig. 1(a) using a single pointer p , as shown in Fig. 1(b), the resulting loop uses only one address register that is allocated to p . The cost of this approach is the cost of a pointer update instruction ($p += 1$) introduced on one of the loop control paths, which is considerably smaller than the cost of spilling/reloading one register. The C code fragment of Fig. 1(b) is a source level model of the *intermediate representation* code resulting after we apply the optimization described in the next sections.

3 Basic Concepts

This section contains a short overview of some basic concepts that are required to formulate the GRA problem and to study its solutions. For a more detailed description, the reader should report to [7].

<pre> (1) for (i = 0; i < N-2; i++) { (2) if (i % 2) { (3) avg += a[i+1] << 2; (4) a[i+2] = avg * 3; (5) } (6) if (avg < error) (7) avg -= a[i+1] - error/2; (8) else (9) avg -= a[i+2] - error; (10) } (11) (12) (13) </pre>	<pre> p = &a[1]; for (i = 0; i < N-2; i++){ if (i % 2) { avg += *p++ << 2; *p-- = avg * 3; } if (avg < error) avg -= *p++ - error/2; else { p += 1; avg -= *p - error; } } </pre>
(a)	(b)

Fig. 1. (a) Code fragment; (b) Modified code that enables the allocation of one register to all references.

First of all assume, for the rest of this paper, that the array data type is a memory word (a typical characteristic of embedded programs). Moreover, assume that each array reference is atomic (i.e. encapsulated into a specific data type in the compiler intermediate representation), and array subscript expressions are not considered for *Common Sub-expression Elimination*(CSE).

A central issue in GRA is to bound allocation to the number of address registers in the target processor. As a consequence, sometimes it is desirable that two references share the same register. This is done by inserting an instruction between the references or by using the auto-increment (decrement) mode. The possibility of using auto-increment (decrement) can be measured by the *indexing distance*, which is basically the distance between two index expressions of two array references.

The concept of live range used in this paper is a straightforward extension of idea of *variable liveness* adopted in the compiler literature [1]. The set of array references of a program can be divided into sets of control-flow paths, each of them defining a live range. For example, in Fig. 2(a) references are divided into live ranges R and S . In our notation for live range, little squares represent array references, and squares with the same color are in the same range. Moreover, an edge between two references of a live range indicates that the references are glued together through auto-increment mode or an update instruction. In Fig. 2 symbol “++” (“--”) following a reference assigns a post-increment (decrement) mode to that reference. Notice that a live range can include references across distinct basic blocks.

As mentioned before, not much research work has been done towards finding solutions to the allocation of address registers for a whole procedure. A solution to this problem has been proposed recently in [7], that is based on repeatedly

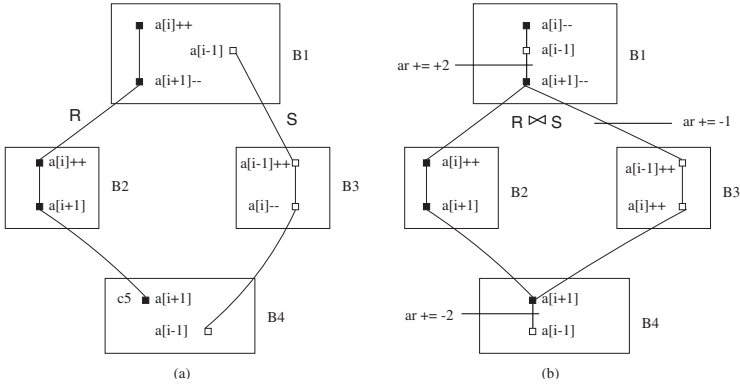


Fig. 2. (a) Two live ranges R and S ; (b) A single live range is formed after merging $R \bowtie S$.

merging pairs of live ranges R and S , such that all references in the new range (called $R \bowtie S$) are allocated to the same register. The algorithm described in [7] starts by partitioning all references across a set of initial live ranges. The initial set of ranges (\mathcal{R}) is formed by the individual references in the loop¹. Ranges are merged pairwise until its number is smaller or equal to the number of address registers available in the processor. This technique is called *Live Range Growth* (LRG). Merge operations for similar problems have also been studied in [14, 9]. The cost of merging two ranges R and S ($cost_{\bowtie}(R, S)$) is the total number of cycles of the update instructions required by the merge. For example, after ranges R and S in Fig. 2(a) are merged, a single range $R \bowtie S$ results (Fig. 2(b)) which requires three update instructions. At each step of the algorithm the pair of ranges of minimum cost is merged.

3.1 The Merge Operator ($R \bowtie S$)

In order to enable references to share the same address register, the LRG algorithm needs to enforce, at compile time, that each reference b should be reached by only one reference a , since this allows the computation of the indexing distance $d(a, b)$ at compile time. As a consequence, the compiler can decide, at compile time, between using auto-increment (decrement) mode for a , or inserting an update instruction on the path from a to b . This requirement can be satisfied if the references in the CFG are in *Single Reference Form* (SRF) [7], a variation of *Static Single Assignment (SSA) Form* [8].

After a program is in SRF, any loop basic block for which array references converge will have a ϕ -equation, including the header and the tail of the loop. After reference analysis, which is performed by solving *reaching definitions* [116] with the data items being the array references, any ϕ -equation has associated to it sets UD and DU , with elements $a_i \in UD$ and $b_j \in DU$. The value w that

¹ Each array access is assigned to a unique range.

results from the ϕ -equation depends on how distant the elements in UD and DU are. We want to select a reference w that reduces the number of update instructions required to merge all a_i to all b_j . This can be done by substituting exhaustively w for all the elements in $UD \cup DU$, and measuring the cost due to the update instructions.

The problem of merging two live ranges can now be formulated as follows. Assume that, during some intermediate step in the LRG algorithm, two ranges R and S are considered for merge. Consider, for the following analysis, only those references which are in $R \cup S$. The set of ϕ -equations associated to these references forms a system of equations. The unknown variables in this system are virtual references w_k . These equations may have circular dependencies, caused basically by the following reasons: (a) it originates from a loop; (b) each ϕ -function depends on its sets UD_ϕ and DU_ϕ . Therefore, any optimal solution for variables w_k cannot always be determined exactly. Consider, for example, the CFG of Fig. 4(a) corresponding to the loop in Fig. 1(a), after ϕ -equations are inserted and reference analysis is performed. Three ϕ -equations result in blocks B_1 , B_3 and B_6 , namely:

$$s1_\phi : w_1 = \phi(w_3, a[i + 1], w_2) \quad (1)$$

$$s2_\phi : w_2 = \phi(a[i + 2], w_1, a[i + 1], a[i + 2]) \quad (2)$$

$$s3_\phi : w_3 = \phi(a[i + 1], a[i + 2], w_1) \quad (3)$$

The work in [7] assumed that the optimal merge of any two ranges is a NP-complete problem. For those instances of the problem in which the exact solution is too expensive, a heuristic should be applied. Any candidate heuristic must choose an evaluation order for the ϕ -equations such that, at each assignment $w_k = \phi(\cdot)$, any argument of ϕ that is a virtual reference is simply ignored. For example, during the estimate of value w_3 , in Equation 3, the argument w_1 is ignored and the resulting assignment becomes $w_3 = \phi(a[i + 1], a[i + 2])$, which results in $w_3 = a[i + 1]$ (or $a[i + 2]$, for the same cost). The final cost of merge is dependent on the order that the heuristic uses to evaluate the set of ϕ -equations. In the heuristic proposed in [7] (called *Tail-Head*), the first equation in the evaluation order is the one at the tail of the loop. From this point on, the heuristic proceeds backward from the tail of the loop up to the header evaluating each equation as it is encountered, and computing the new value of w_k , which is then used in the evaluation of future equations.

Example 1. Consider the application of this heuristic to the Equations 1-3, from the code fragment in Fig. 1(a). The result is shown in Fig. 3(a). First, the ϕ -equation $s3_\phi$ at the tail of the loop (block B_6) evaluates to $w_3 = a[i + 1]$. This reference is equivalent to $a[i]$ in the next loop iteration. Next, $s2_\phi$ and $s1_\phi$ are evaluated, resulting in $w_2 = a[i + 2]$ and $w_1 = a[i + 1]$, respectively. At this point, all virtual references (i.e. the values of w_k) have been computed. In the last step, update instructions and auto-increment(decrement) modes are inserted into the code such that the references associated to the live ranges being merged satisfy the computed ϕ -functions. This results in the addition of auto-decrement mode

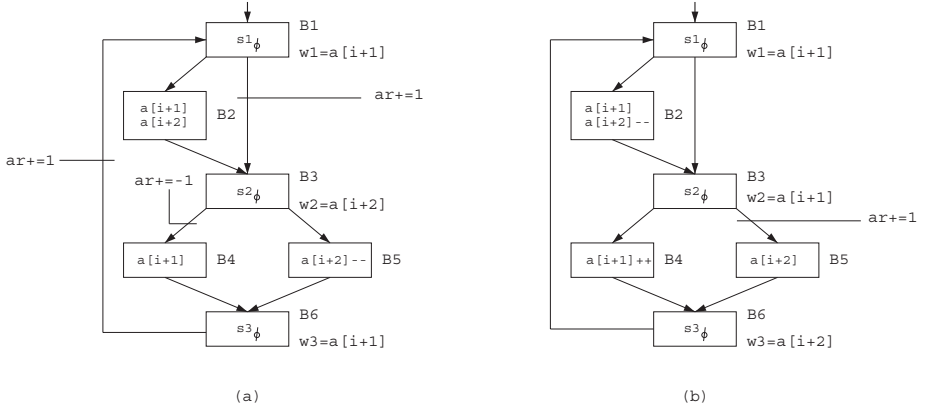


Fig. 3. (a) Mode and update-instruction insertion for Tail-Head heuristic; (b) Optimal mode and update-instruction insertion using dynamic programming.

to $a[i+2]$ in B_5 , and the insertion of three update instructions: (a) $ar+ = 1$ on the edge (B_1, B_3) ; (b) $ar+ = 1$ on the edge (B_6, B_1) ; and (c) $ar+ = -1$ on the edge (B_3, B_4) .

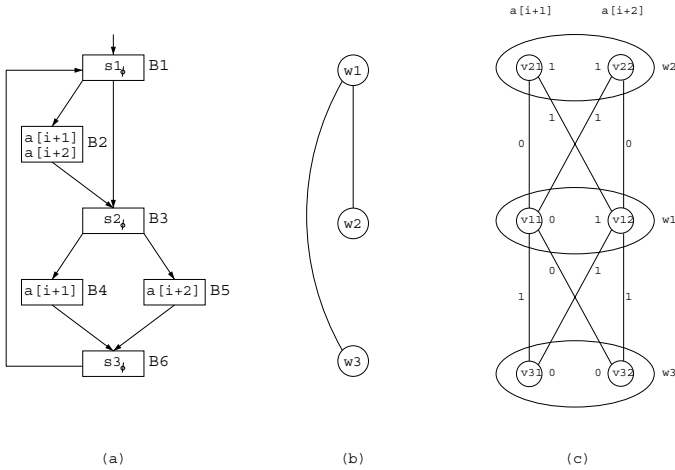


Fig. 4. (a) The CFG in SRF; (b) The DG_ϕ representation; (c) The SG_ϕ for the DG_ϕ .

4 Complexity Analysis of $R \bowtie S$

Given that the merge operation is central to the LRG algorithm, in the following sections we perform the analysis of this operation. We complete the work started in [7] by proving that the merge operation (i.e., finding the minimal cost for it) is an NP-hard problem in general, through a reduction from the minimal vertex-covering problem. Moreover, we show that in a special case the merge operation admits an optimal solution, based on a dynamic programming algorithm. For the other cases, we show that a simple heuristic leads to effective solutions.

4.1 The Φ -Dependence Graph

The problem of solving the set of equations above can be formulated as a graph theoretical one as follows. Consider the code fragment of Fig. 4(a). We build a dependence graph where vertices are associated to the ϕ -equations and edges define dependencies between equations. This graph is called ϕ -Dependence Graph (DG_ϕ) and is constructed as follows:

Definition 1. *The DG_ϕ is an undirected graph for which there exist a vertex associated to each ϕ -equation, and there exist an edge (w_i, w_j) , between two vertices w_i and w_j , if and only if w_i is a ϕ -function of w_j and vice-versa.*

Figure 4(b) shows the DG_ϕ corresponding to the equations in Fig. 4(a).

4.2 The Φ -Solution Graph

Having constructed the DG_ϕ for a set of ϕ -equations, we build a graph which reflects all possible solutions for these equations. But before doing so, we need to define some concepts. Let m be the number of distinct references that can result from the ϕ -equations. The range of references can be obtained by a simple code inspection, searching for the minimum and maximum values for each reference in the loop², which are stored into vector $\text{refs}[i]$, $1 \leq i \leq m$. Using these concepts we can now construct a graph that encodes all the possible solutions for the ϕ -equations.

Definition 2. *The Φ -Solution Graph (SG_ϕ) is an undirected graph constructed from the DG_ϕ as follows: (1) For each vertex u in DG_ϕ insert m vertices into SG_ϕ , one for each possible solution of the equation associated to u ; (2) If there is an edge in DG_ϕ between two vertices u and v , and being sets $\{u_1, u_2, \dots, u_m\}$ and $\{v_1, v_2, \dots, v_m\}$ the vertices in the SG_ϕ associated with u and v respectively, insert edges (u_i, v_j) into SG_ϕ , for all $1 \leq i, j \leq m$.*

Figure 4(c) shows the SG_ϕ obtained from the DG_ϕ in Fig. 4(b). Notice that SG_ϕ is a multipartite graph, i.e. $SG_\phi = \{P_1, P_2, \dots, P_n\}$, with $P_i = \{v_{ij}, | 1 \leq j \leq m\}$. SG_ϕ is a weighted graph, with costs both on its vertices and edges. We

² It may be necessary to consider the loop increment.

assign a local cost $lcost(v_{ij})$ to each vertex v_{ij} , where $1 \leq i \leq n$ and $1 \leq j \leq m$, and define it as follows:

$$lcost(v_{ij}) = \sum_{j=1}^m cost(refs[j], a) , \quad (4)$$

for each real reference a in $args[w_i]$, where $args[w_i]$ is the set of all arguments of the ϕ -function associated with w_i . The local cost is computed using the same cost function defined in [7] (which considers the possibility of using the auto-increment(decrement) modes), by assuming that $w_i = refs[j]$ and considering only the arguments of ϕ_i which are real references (i.e. not w). In other words, $lcost$ gives the contribution to the total merge cost when the solution for the ϕ -equation associated to w_i is $w_i = refs[j]$. We also assign to each edge $e = (v_{ij}, v_{kl})$ in SG_ϕ , a global cost given by

$$gcost(v_{ij}, v_{kl}) = cost(refs[j], refs[l]) . \quad (5)$$

Notice that both arguments of $gcost$ are virtual references, and reflect the cost of assigning solutions $refs[j]$ and $refs[l]$ respectively to virtual references w_i and w_k . Figure 4(c) illustrates the local and global costs associated with each SG_ϕ vertex and edge.

Now we are able to state the solution of the ϕ -equation system in terms of the SG_ϕ . We want to determine the set $V = \{v_i | v_i \in P_i, 1 \leq i \leq n\}$ such that the total merge cost $cost_\bowtie[V] = cost_\bowtie(R, S)$ is minimum, where

$$cost_\bowtie[V] = \sum lcost(v_i) + \sum gcost(e) , \quad (6)$$

for all $1 \leq i \leq n$ and $e \in SG_\phi[V]$, where $SG_\phi[V]$ denotes the sub-graph of SG_ϕ induced by the vertices of V .

4.3 $R \bowtie S$ is NP-hard

In this subsection, we prove that the problem stated above is NP-hard. To do this, we re-state it as a decision problem, and call it the *Minimum Cost Induced Sub-graph Problem (MCISP)*.

Definition 3 (MCISP). *Let G be an instance of a SG_ϕ and k a positive integer. Find a subset V of the vertices of G (as described in Sect. 4.2) such that $cost_\bowtie[V] \leq k$.*

We show that MCISP is NP-hard through a reduction from the *Vertices Covering Problem (VCP)* [10], defined as follows.

Definition 4 (VCP). *Given a graph G and an integer k , decide whether G has a vertex-cover C with at most k vertices, such that each edge of G has at least one of its incident vertices in C .*

Theorem 1. *MCISP is NP-complete.*

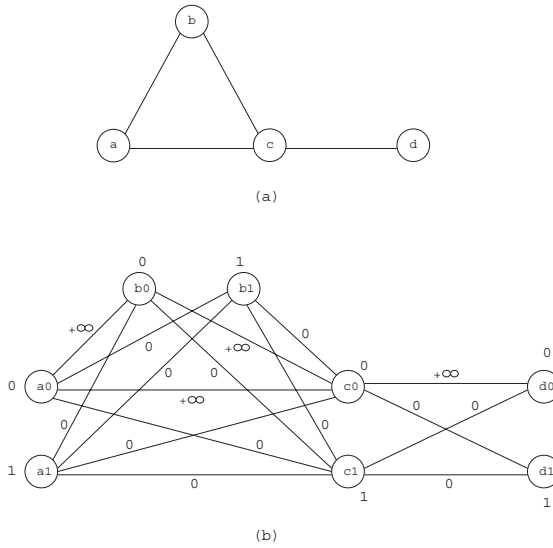


Fig. 5. (a) VCP on original instance graph G ; (b) MCISP on reduced instance graph G' .

Proof. First, it must be noticed that $\text{MCISP} \in \text{NP}$. This is an easy task, as we can verify in polynomial-time if a given solution V has a total cost less than or equal to k . We will not give more details on this.

Now, for MCISP to be NP-complete, it is missing to show that it is NP-hard. To do this, we use a reduction from a known NP-complete problem to MCISP. We choose the vertex-cover problem (VCP) for this purpose.

Consider now the reduction of VCP, with instance graph G and integer k , to MCISP, with G' and k' . First we show how to create G' from G . For example, consider the graph G in Fig. 5(a). For each vertex $v \in G$, we insert two vertices v_0 and v_1 in G' . Then, for each edge $e = (u, v) \in G$, we insert four edges in G' : (u_0, v_0) , (u_0, v_1) , (u_1, v_0) and (u_1, v_1) .

Next, we add the costs to G' , both to its vertices and edges, to reflect the local and global costs defined above. For each v_0 we make $\text{lcost}(v_0) = 0$, and for each vertex v_1 we set $\text{lcost}(v_1) = 1$.

On the edges, the following costs are assigned: $\text{gcost}(u_0, v_0) = +\infty$, and $\text{gcost}(u_0, v_1) = \text{gcost}(u_1, v_0) = \text{gcost}(u_1, v_1) = 0$. Fig. 5(b) illustrates the graph G' obtained from G in Fig. 5(a). It is clear that this reduction can be done in polynomial-time. Now we are going to prove that, having obtained G' from G as described above, and setting $k = k'$, VCP on G and k is satisfied if and only if MCISP on G' and k' is satisfied. Let's first see why the forward argument holds. Let C be a solution to VCP. So, for each edge in G , at least one of its incident vertices is in C . We argue that $V = V_1 \cup V_0$, where $V_1 = \{v_1 | v \in C\}$ and $V_0 = \{v_0 | v \notin C\}$ is a solution to MCISP. Let $q \leq k$ be the cost of C (i.e., the number of vertices). Consider now the cost of the solution V to MCISP. The

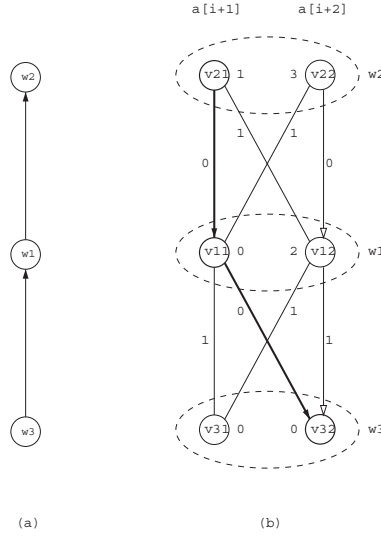


Fig. 6. (a) The DDG_ϕ for the DG_ϕ of Fig. 4; (b) The DSG_ϕ for the DDG_ϕ in (a).

first summation on Equation 6 is exactly the number of vertices in V_1 , which is also the number of vertices in C (i.e. q). Now consider the second summation in Equation 6. As C is a solution to VCP, there are no two vertices $u_0, v_0 \in V$ such that the edge $(u_0, v_0) \in G'[V]$. Hence, every edge in $G'[V]$ has its $gcost$ equal to 0, and $cost_M[V]$ is exactly q .

We still have to prove the backward argument, i.e., that given a solution V to MCISP with cost $q \leq k'$, there is a solution C to VCP with the same cost q . Being k' a finite integer number, $G'[V]$ has no edge between any two vertices u_0 and v_0 . Therefore, choosing $C = \{v | v_1 \in V\}$, any edge in G will have at least one of its incident vertices in C , and so C is a vertex-cover for G .

Now it is just missing to show that the cost of solution C to VCP is also q . This is easy to see, as the cost q is exactly the number of vertices in V with subscript 1, which is also the number of vertices in C . And so, the correctness of the polynomial-time reduction is proved.

Having proved the decision version of the problem to be NP-complete, and being the problem an optimization one, we conclude that the problem of finding the minimal cost for the merge operation is NP-hard.

5 The Case when DG_ϕ Is a Tree

As shown in the previous section, the merge operation is NP-hard. In this section we present an optimal exact algorithm for merging two live ranges, under the assumption that the corresponding DG_ϕ is a tree. The algorithm is based on the dynamic programming technique, and uses a special order to traverse the nodes

of the DG_ϕ . Ordering the traversal of the vertices actually means imposing a direction on each edge of the dependence graph. The arc $(u \rightarrow v)$ obtained by choosing a direction for the edge (u, v) implies that w_u is an argument of w_v , unlike in the previous form of the DG_ϕ , when edge (u, v) indicated that w_v was also an argument of w_u . In this directed form of the DG_ϕ (called DDG_ϕ) the other arguments of the ϕ -functions are not changed.

Our algorithm is based on a particular traversal of the DDG_ϕ that we call *Leaves Removal Order* (LRO). This order is obtained by successively removing any leaf of the tree. If l is a leaf, it has a unique adjacent node, say v , such that the direction of (l, v) is set to $l \rightarrow v$. When l is removed, w_v is dropped out of the list of arguments of the ϕ -function for w_l .

This ordering algorithm can be implemented with time-complexity $O(n)^3$, for example by taking into account the degree of each vertex.

Figure 6(a) shows the DDG_ϕ obtained from the DG_ϕ of Fig. 4(a), using the algorithm just described above. We call the attention to the fact that, although we described this ordering algorithm separately, in practice it can be implemented on the same pass as the next step.

The next step in our solution is a dynamic programming algorithm which computes the minimum merge cost for any solution of our problem. We call this algorithm the *LRO Algorithm*, and show its pseudo-code in Alg. 1. We need a vector $accost[1, \dots, m]$ for each vertex of DDG_ϕ , to hold the cumulative cost at this vertex, for each one of the m possible solutions of the corresponding ϕ -equation, from $refs[1]$ to $refs[m]$, respectively. In other words, for each vertex v of the DG_ϕ we assign an element $accost[i]$ to each one of its corresponding v_i vertices in SG_ϕ . The $accost$ vectors are initialized with the respective $lcost$ vectors. Then, following the LRO, for each leaf node l being removed, such that $l \rightarrow v$ is its last arc, we add the cost c_{v_i} to $accost_v[i]$, where c_{v_i} is the minimum value between $accost_l[j] + gcost(v_i, l_j)$, for all $1 \leq j \leq m$.

Algorithm 1 LRO algorithm

- (1) procedure $LRO(DG_\phi, SG_\phi)$
 - (2) for each vertex $v_{ij} \in SG_\phi$, where $1 \leq i \leq n$ and $1 \leq j \leq m$, do
 - (3) $accost_i[j] \leftarrow lcost(v_{ij})$
 - (4) for all but the last vertex $v_p \in DG_\phi$, according to LRO, do
 - (5) let (v_p, v_q) be the last edge incident to v_p
 - (6) for $j \leftarrow 1$ to m do
 - (7) $min \leftarrow +\infty$
 - (8) for $i \leftarrow 1$ to m do
 - (9) if $accost_p[i] + gcost(v_{pi}, v_{qj}) < min$ then
 - (10) $min \leftarrow accost_p[i] + gcost(v_{pi}, v_{qj})$
 - (11) $accost_q[j] \leftarrow accost_q[j] + min$
-

³ Remember that n is the number of vertices of DG_ϕ

The minimum total merge cost is the minimum value in the DG_ϕ last vertex's $accost$ vector. For simplicity, we omitted from Alg. [1](#) the code needed to recover the solution (i.e. the set $S = \{v_i | v_i \in P_i, 1 \leq i \leq n\}$), though it should be clear that it can be implemented without any extra computational complexity cost. The time-complexity of this algorithm is $O(n.m^2)$. In practice, since m is generally bounded by a small constant⁴, the algorithm complexity reduces to $O(n)$.

Lemma 1. *Given a LRO sequence $S = (v_1, v_2, \dots, v_n)$ of DG_ϕ 's vertices, the LRO Algorithm properly computes the minimal value for $accost[v_{ij}]$, for every $1 \leq i \leq n$ and $1 \leq j \leq m$.*

Proof. First, it must be noticed that, as long as DG_ϕ is a tree, there always exists an LRO. Moreover, the ϕ -functions are inserted such that the cost of every arc in the CFG affected by the ϕ -functions' values is considered exactly once. Let $S_k = (v_1, \dots, v_k)$. We prove this lemma by induction in k .

Basis: $k = 1$. In this case, $accost[v_{1j}]$, $1 \leq j \leq m$, is just equal to $lcost(v_{1j})$, which is minimal, as all the $lcost$'s are chosen to be locally optimal.

Induction Hypothesis: We assume that, for every $i < k$, the LRO Algorithm computes the minimal values for $accost[v_{ij}]$.

Inductive Step: We define the set $A = \{v_i | i < k \text{ and } (v_i, v_k) \in DG_\phi\}$. We now need to show how to compute the minimal value for each $accost[v_{kj}]$, $1 \leq j \leq m$. The $lcost(v_{kj})$ is always in $accost[v_{kj}]$. So we have to minimize

$$\sum_{v_i \in A} \min\{accost[v_{iy}] + gcost(v_{kj}, v_{iy}) | 1 \leq y \leq m\}$$

for each $1 \leq j \leq m$. $gcost(v_{kj}, v_{iy})$ is a precalculated constant. By the induction hypothesis, $accost[v_{iy}]$ is minimal, for each $v_i \in A$ and $1 \leq y \leq m$. So, it turns out that $accost[v_{kj}]$ is correctly computed, for each $1 \leq j \leq m$, as we minimize a value among the minimal values for all possible choices. And so the proof is complete, and the LRO Algorithm computes the minimal values for $accost[v_{kj}]$, for all valid values for k and j .

Theorem 2. *The LRO Algorithm is optimal when the DG_ϕ is a tree, in the sense that it results in the smallest possible cost for $R \bowtie S$.*

Proof. The Lemma [1](#) states that the LRO Algorithm computes the minimum cost for $accost[v_{ij}]$, for all vertices $v_{ij} \in SG_\phi$, if the corresponding DG_ϕ is a tree. So, for the algorithm to compute the smallest cost for $R \bowtie S$, we just have to take the minimum between $\{accost[v_{nj}] | v_{nj} \in SG_\phi\}$, and the v_n is the last vertex in the LRO for DG_ϕ .

⁴ Remember that m is the number of references (inside the loop) in the range defined by the one with the minimum and the one with the maximum index value.

Example 2. Figure 6(b) shows the result of running this algorithm on the SG_ϕ from Fig. 4(c), according to the LRO from Fig. 6(a). In this figure, we show the global costs on the edges, and the final cumulative cost on each vertex. At each vertex v , we use arrows to point to the other vertices u (a single vertex in this example), such that u is used to reach the minimum value at v . The vertices and edges that compose the solution are highlighted in Fig. 6(b).

Figure 3(b) shows the code fragment of Fig. 1(a) with the modifications associated to the solution illustrated in Fig. 6(b). Notice that only one update instruction is necessary, which is in accordance with $cost[v_{21}]$ in Fig. 6(b). Figure 3(a) shows the solution found using the Tail-Head heuristic described in 7, for the instance of the problem shown in Fig. 1(a). This solution has a higher cost (3 update instructions).

6 Experimental Results

Our experimental framework is based on programs from the MediaBench benchmark [18], which are typically found in many embedded applications. We have implemented the LRG heuristic 7, and the LRO algorithm described in Sect. 5 into the IMPACT compiler [17]. Both algorithms run at `lcode` level, just before all standard intermediate representation optimizations [1]. At each merge in the LRG algorithm, the DG_ϕ graph is tested for the presence of cycles. If DG_ϕ is cyclic the Tail-Head (TH) heuristic is applied. On the other hand, if DG_ϕ is acyclic, the LRO algorithm is used to compute the ϕ -functions.

Table 1. Comparison between the Tail-Head (TH) and LRO+TH approaches.

Program Name	Loop	Minimum No. ARs	% Trees	# Update Instr.	
				TH	LRO+TH
jpeg	1	2	100	4	3
	2	1	64	4	3
	3	2	46	4	4
epic	1	1	78	0	0
	2	1	100	3	3
	3	2	77	2	1
pgp	1	2	100	0	0
	2	1	100	1	1
mpeg	1	1	75	3	3
mesa	1	1	75	3	3
	2	1	100	2	2
pegwit	1	1	67	2	2
	2	2	100	1	0
gsm	1	2	100	0	0
	2	1	89	1	1

For each program, we measured the number of update instructions required by its inner loops [\[8\]](#), such that one address register is used to access all array references of each array (Table [II](#)). This is a worst case scenario which reveals the improvement resulting from the LRO algorithm. This table also shows an estimate of the number of times when DG_ϕ is a tree during the live range growth process. These preliminary results indicate that this may be a frequent situation in embedded applications. We acknowledge that measuring the final cycle count would be more realistic, but unfortunately at this point we do not have a complete retarget of the IMPACT architecture to a real-world DSP processor. On the other hand, DSP compilers have few address registers available and some of them are used to perform other tasks (e.g. stack access), such that only very few are left for address register allocation. In this case, the above scenario approaches what occurs in many real situations. Notice from Table [II](#), that the improvement due to the LRO algorithm is typically one less update instruction in the loop body, what considerably improves loop performance, particularly for tight inner loops found in many critical signal processing applications.

Even more relevant, these numbers also reveal that the TH heuristic that we proposed in [\[7\]](#) results in near optimal allocation.

7 Conclusions and Future Work

This paper addresses the problem of allocating address registers to array references in loops of embedded programs. It is based on a previous work that assigns address registers to live ranges by merging ranges together. The contributions of this work are two. First, it proves that the optimal merge of address register live ranges is an NP-hard problem. Second, it proves the existence of an optimal linear-time algorithm to merge live ranges when the dependency graph formed by the set of ϕ -equations on the references is a tree.

A full evaluation of this technique on a real-world DSP running a wide set of benchmarks is under way. We are also working to add support to new DSP features. One example is the support to modify registers. This kind of register is present in many DSPs and we intend to use them to increment address registers by values greater than one, incurring in no additional cost. By doing so, we can avoid issuing some update instructions and, as a consequence, decrease the update instruction overhead.

Acknowledgments. This work was partially supported by CNPq (300156/97-9), ProTem CNPq/NSF Collaborative Research Project (68.0059/99-7) and by fellowship grants CNPq (141710/2000-4) and (141958/2000-6). We also thank the reviewers for their comments.

⁵ Only those loops which make access to arrays have been considered.

References

1. AHO, A., SETHI, R., AND ULLMAN, J. *Compilers, Principles, Techniques and Tools*. Addison Wesley, Boston, 1986.
2. ANALOG DEVICES. *ADSP-2100 Family User's Manual*.
3. ARAUJO, G., SUDARSANAM, A., AND MALIK, S. Instruction set design and optimizations for address computation in DSP processors. In *9th International Symposium on Systems Synthesis* (November 1996), IEEE, pp. 31–37.
4. BRIGGS, P., COOPER, K., KENNEDY, K., AND TORCZON, L. Coloring heuristics for register allocation. In *Proc. of the ACM SIGPLAN'89 on Conference on Programming Language Design and Implementation* (June 1982), pp. 98–105.
5. CHAITIN, G. Register allocation and spilling via graph coloring. In *Proc. of the ACM SIGPLAN'82 Symposium on Compiler Construction* (June 1982), pp. 98–105.
6. CHOW, F., AND HENNESSY, J. L. The priority-based coloring approach to register allocation. *ACM Trans. Program. Lang. Syst.* 12, 4 (October 1990), 501–536.
7. CINTRA, M., AND ARAUJO, G. Array reference allocation using SSA-Form and live range growth. In *Proceedings of the ACM SIGPLAN LCTES 2000* (June 2000), pp. 26–33.
8. CYTRON, R., FERRANTE, J., ROSEN, B., WEGMAN, M., AND ZADECK, F. An efficient method of computing static single assignment form. In *Proc. of the ACM POPL'89* (1989), pp. 23–25.
9. ECKSTEIN, E., AND KRALL, A. Minimizing cost of local variables access for DSP-processors. In *LCTES'99 Workshop on Languages, Compilers and Tools for Embedded Systems* (Atlanta, July 1999), Y. A. Liu and R. Wilhelm, Eds., vol. 34(7) of *SIGPLAN*, ACM, pp. 20–27.
10. GAREY, M., AND JOHNSON, D. *Computers and Intractability*. W. H. Freeman and Company, New York, 1979.
11. GEBOTYS, C. DSP address optimization using a minimum cost circulation technique. In *Proceedings of the International Conference on Computer-Aided Design* (November 1997), IEEE, pp. 100–103.
12. GUPTA, R., SOFFA, M., AND OMBRES, D. Efficient register allocation via coloring using clique separators. *ACM Trans. Programming Language and Systems* 16, 3 (May 1994), 370–386.
13. LEUPERS, R., BASU, A., AND MARWEDEL, P. Optimized array index computation in DSP programs. In *Proceedings of the ASP-DAC* (February 1998), IEEE.
14. LIAO, S., DEVADAS, S., KEUTZER, K., TJANG, S., AND WANG, A. Storage assignment to decrease code size. *ACM Transactions on Programming Languages and Systems* 18, 3 (1996), 235–253.
15. MOTOROLA. *DSP56000/DSP56001 Digital Signal Processor User's Manual*, 1990.
16. MUCHNICK, S. S. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.
17. POHUA P. CHANG SCOTT A. MAHLKE WILLIAM Y. CHEN, N. J. W., AND MEI W. HWU, W. Impact: An architectural framework for multiple-instruction-issue processors. 266–275.
18. POTKONJAK, C. L. M., AND MANGIONE-SMITH, W. H. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems.

Speculative Prefetching of *Induction Pointers*

Artour Stoutchinin¹, José Nelson Amaral², Guang R. Gao³, James C. Dehnert⁴, Suneel Jain⁵, and Alban Douillet³

¹ STMicroelectronics, Grenoble, France

² Department of Computing Science, University of Alberta,
Edmonton, AB, T6G-2E8, Canada

amaral@cs.ualberta.ca, <http://www.cs.ualberta.ca/~amaral>

³ Computer Architecture and Parallel System Laboratory,
University of Delaware, Newark, DE, USA

{ggao, douillet}@capsl.udel.edu, <http://www.capsl.udel.edu>

⁴ Transmeta Co., Santa Clara, CA, USA, dehnert@transmeta.com

⁵ Hewlett-Packard Co., Cupertino, CA, USA, sjain@cup.hp.com

⁶ Stoutchinin, Dehnert, and Jain were at SGI when most of this research was conducted.

Abstract. We present an automatic approach for prefetching data for linked list data structures. The main idea is based on the observation that linked list elements are frequently allocated at constant distance from one another in the heap. When linked lists are traversed, a regular pattern of memory accesses with constant stride emerges. This regularity in the memory footprint of linked lists enables the development of a prefetching framework where the address of the element accessed in one of the future iterations of the loop is dynamically predicted based on its previous regular behavior.

We automatically identify pointer-chasing recurrences in loops that access linked lists. This identification uses a surprisingly simple method that looks for *induction pointers* — pointers that are updated in each loop iteration by a load with a constant offset. We integrate induction pointer prefetching with loop scheduling. A key intuition incorporated in our framework is to insert prefetches only if there are processor resources and memory bandwidth available. In order to estimate available memory bandwidth we calculate the number of potential cache misses in one loop iteration. Our estimation algorithm is based on an application of graph coloring on a memory access interference graph derived from the control flow graph. We implemented the prefetching framework in an industry-strength production compiler, and performed experiments on ten benchmark programs with linked lists. We observed performance improvements between 15% and 35% in three of them.

1 Introduction

Modern computers feature a deep memory hierarchy. The data move from main memory to processor register files through a number of levels of caches. Prefetch

<pre> 1 max = 0; 2 current = head; 3 while(current != NULL) { 4 if(current -> key > max) 5 max = current -> key; 6 current = current -> next; 7 }</pre>	<pre> 1 max = 0; 2 current = head; tmp = current; 3 while(current != NULL) { 4 if(current -> key > max) 5 max = current -> key; 6 current = current -> next; stride = current - tmp; prefetch(current + stride*k); tmp = current; 7 }</pre>
--	--

Fig. 1. A pointer-chasing loop that scans a linked list without and with prefetching.

mechanisms can bring data into the caches before it is referenced. These mechanisms rely on the spatial and temporal locality of data that naturally occurs in regular programs. However, for irregular, pointer-based applications, prefetching is often less effective. Some critical issues that make the prefetching of pointer-based data more difficult include:

1. Pointer-based applications display poor spatial and temporal locality. Unlike array elements, consecutive data elements of a pointer-linked data structure do not necessarily reside in adjacent memory locations. Also, references to many unrelated data locations often intervene between reuses of linked list elements.
2. In general, the address of a data element accessed in iteration i is not known until iteration $i - 1$ is executed, and thus cannot be prefetched in advance. This address dependence is known as the *pointer-chasing* problem.
3. Most pointer-based data structures are allocated in heap memory. Because the heap is typically a large memory space, and these structures can be anywhere in that space, heap allocation makes it difficult for compilers to reliably predict cache misses or perform reuse analysis. Ad-hoc generation of prefetches may result in performance degradation due to increased memory traffic and cache pollution.

Prefetching in conjunction with software pipelining has been successfully used in optimization of numerical loops [12]. In this work we perform *induction pointer prefetching* to optimize pointer chasing loops. Our objective is to use induction pointer prefetching to improve the software pipelining initiation interval and to prevent the software pipeline from being disrupted because of primary cache misses. Usually, the scheduler builds the software pipeline optimistically assuming cache hit load latencies. If a load misses in the cache, the pipeline is disrupted and loses its efficiency.

The induction pointer prefetching identifies *induction pointers* in pointer-chasing loops and generates code to compute their memory access stride and to

prefetch data. A loop contains a recurrence if an operation in the loop has a direct or indirect dependence upon the same operation from some previous iteration. The existence of a recurrence manifests itself as a cycle in the dependence graph. A *pointer-chasing recurrence* is a recurrence that only involves loads with constant offset and copy operations. We call the addresses used by loads involved in the pointer-chasing recurrences *induction pointers*. For instance, in the example in Figure 1, **current** is an induction pointer. The term comes from the fact that such loads are similar to induction variables in that the address for the load in the next iteration is provided by the execution of the load in the current iteration.

Our prefetch method is motivated by the observation that, although the elements of a linked list are not necessarily stored contiguously in memory, there tends to be a regularity in the allocation of memory for linked lists. If the compiler is able to detect a linked list scanning loop, it can assume that the accesses to the list elements generate a regular memory reference pattern. In this case, the address of an element accessed in iteration $i + k$ is likely to be the effective address of the element accessed in iteration i plus $k * S$, where S is the constant address stride. The compiler can then insert code that dynamically computes the stride S for induction pointers. Figure 1 shows a pointer chasing loop that scans a linked list before and after prefetch insertion. See [19] for the actual intermediate code used for prefetching.

Once the memory access stride of induction pointers is identified, data whose addresses are computed relative to the induction pointers are speculatively prefetched. The prefetch instructions issued do not cause exceptions and even if their address is mispredicted the program correctness is preserved. We perform a *profitability analysis* to establish whether prefetching is beneficial. Prefetching is only performed if the analysis indicates that there are unused processor resources and memory bandwidth for it. Our profitability analysis compares the upper bounds on the initiation rate imposed by data dependency constraints and by processor resource constraints. To prevent processor stalls due to prefetching, we also estimate the number of outstanding cache misses in any iteration of the loop. We avoid prefetching when this number exceeds the maximum number supported by the processor.

This paper addresses the set of open questions listed below. Although the problem of prefetching induction pointers, with and without hardware support, has been studied before (see Section 5), to the best of our knowledge the framework presented in this paper is the first that relies exclusively on compiler technology:

- How to identify pointer-chasing recurrences using a low complexity algorithm? (see Section 2.1)
- How to decide when there are enough processor resources and available memory bandwidth to profitably prefetch an induction pointer? (see Section 2.5)
- How to efficiently integrate induction pointer prefetching with loop scheduling based on the above profitability analysis? Such integration becomes complex for pointer chasing loops with arbitrary control flow. (see Section 3)

- How to formulate, algorithmically, the problem of maximizing the use of cache-memory bandwidth in non-blocking cache systems that support multiple outstanding cache misses? (see Section 2.4)
- How well does speculative prefetching of induction pointers work on an industry-strength state-of-the-art optimizing compiler? (see Section 4)

We describe our profitability analysis in section 2 and the prefetch algorithm in section 3. We present the wall-clock execution time measurements and the variation in the number of cache misses and TLB misses incurred with and without prefetching in section 4. Finally we discuss related research in section 5.

2 Induction Pointer Identification and Profitability Analysis

Ideally, we would like speculative prefetching to never cause any performance degradation. Induction pointer prefetching may lead to performance degradation as a result of one of the following: increased processor resource requirements (computation units, issue slots, register file ports, etc); increased memory traffic due to potential fetching of useless data; increased instruction cache misses; and potential displacement of useful data from the cache.

Our profitability analysis addresses the problems of not degrading performance due to the increase in required processor resources and in the memory traffic. In our experience, pointer chasing loops are short, therefore instruction cache misses are not an important concern. In our current implementation, the profitability analysis does not take into account the cache pollution problem. Nonetheless the effects of cache pollution are reflected in the results presented in Section 4.

The execution time of a loop is determined by the rate at which iterations can be initiated. We attempt to avoid performance degradation by estimating the constraints imposed by processor resources and by recurrent data dependencies on the initiation rate of each loop. Based on these constraints we decide when we should prefetch for a given induction pointer.

The *initiation interval* (II) of a loop with a single control path is defined as the number of clock cycles that separate initiations of consecutive iterations of the loop. The minimum initiation interval (MII) defines an upper bound on the initiation rate of instructions in a loop. The MII is computed as the maximum of *recurrence MII* and the *resource MII* [5].

In loops with multiple control paths, the interval between consecutive executions of any given operation depends on the control path executed in each iteration, and may change from iteration to iteration. A conservative upper bound on the initiation rate of an operation L corresponds to the maximal MII of all control paths that execute L .

We base our decision to prefetch data for a load instruction that belongs to an induction pointer recurrence on three estimates: (1) an estimate of the conservative upper bound on its initiation rate; (2) an estimate of the potential

node = ptr->next; ptr = node->ptr; (a)	r1 <- load r2, offset_next r2 <- load r1, offset_ptr (b)
father = father->pred (c)	r2 <- r1 r1 <- load r2, offset_pred (d)

Fig. 2. Examples of code occurring in pointer-chasing loops.

increase in cache misses; and (3) an estimate of the potential resource usage increase. Prefetching is deemed profitable if (i) it does not decrease the conservative upper bound on load's initiation rate, and (ii) a potential extra cache miss caused by the prefetch does not result in a stall of the cache fetching mechanism.

2.1 Identification of List Traversing Circuits

We identify circuits in the data dependence graph of a loop that are indicative of a linked list being traversed. Consider the code statements shown on Figure 2 (a) and (c) that are often used to traverse linked lists. These statements are translated by the SGI compiler into the corresponding assembly sequences shown on Figure 2 (b) and (d). Our key observation is that a circuit formed exclusively by loads (with some offset) and copy instructions is likely to be a pointer-chasing circuit.

We used the two patterns shown in Figure 2 for identification of the pointer-chasing circuits. Conceptually our algorithm finds all the elementary circuits in the loop's data dependence graph using Tarjan's algorithm [20] and then finds those circuits consisting exclusively of loads with constant offsets and copy instructions. Enumerating all circuits can be exponential in a general graph. Our implementation avoids enumeration by identifying only the pointer-chasing circuits. This has one important implication for the efficiency of our implementation: Tarjan's algorithm's complexity is $O(N * E * C)$, where N and E are respectively the number of nodes and the number of edges in the data dependence graph, and C is the number of elementary circuits. Since in practice loops tend to have very few pointer-chasing circuits (one in most cases), our implementation practically achieves $O(N * E)$ complexity.

The initiation rate of operations in list traversing circuits is bounded by induction pointer recurrences and the resource usage. A precise determination of the such bound should match each control path with recurrences that it executes. However, such matching is expensive. Instead, we consider that the initiation rate of an operation is limited by the most resource constrained control path in which this operation executes, and by the most constraining recurrence to which this operation belongs.

2.2 Recurrence Bound on the Initiation Rate

The recurrence minimum initiation interval (*recMII*) of a loop is a lower bound imposed on the MII by recurrences (cyclic data dependences) among operations from multiple iterations of the loop. In classical software pipelining the *recMII* for a recurrence c is given by the quotient of the sum of the latencies of the operations in c , $latency(c)$ and the sum of the *iteration distances* of the operations in c , $iteration\ distance(c)$ [5,15]. The iteration distance of each dependence in c is equal to the number of iterations separating the dependent operations.

$$recMII(c) = \left\lceil \frac{latency(c)}{iteration\ distance(c)} \right\rceil$$

When a loop L has a single control path, the *recMII* for the loop is computed by taking the maximum of the *recMII*(c) over all the elementary circuits in the data dependence graph of the loop.

In this framework we are also interested in prefetching in loops that have multiple control paths. Therefore we must extend the concept of *recMII* to suit our purposes. To this end we define the *recMII* associated with each instruction L in the loop, *recMII*(L). Instead of all recurrences in the loop, the *recMII*(L) considers only the recurrences in which L participates. Therefore *recMII*(L) is the maximum of *recMII*(c) among all recurrences that execute L , and defines a *conservative bound* on the initiation rate of instruction L .

$$recMII(L) = \max_{c|L \in c} \{recMII(c)\}$$

When computing $latency(c)$ we assume that the loads in a pointer chasing recurrence incur cache misses. As prefetches are added, the prefetched loads are optimistically upgraded to cache hits, and the value of $latency(c)$ is recomputed. Therefore prefetching a load L reduces the *recMII*(c) for all the recurrences that contain the load L .

2.3 Resource Bound on the Initiation Rate

The resource usage of the operations along a control path of a loop imposes another upper bound on the rate at which operations in that path can be initiated, the resource minimal initiation interval (*resMII*). The usage of each resource in a control path is calculated by adding the resource usage of all operations in that path. A conservative bound on the initiation rate of operation L in basic block B is the maximum over lower bounds imposed by resource usage in each control path that executes B :

$$resMII(L) = resMII(B) = \max_{p|B \in p} \{resMII(p)\}, \quad L \in B$$

where $resMII(p)$ is the resource imposed lower bound on the instruction initiation rate in the control path p .

The $resMII(B)$ for each basic block is computed using the DAG longest path algorithm [4] on the control flow graph (CFG) [1]. Each vertex in the CFG represents a basic block B . We associate a vector of weights, $w(B)$ to each basic block. Each element of $w(B)$ represents the usage of a processor resource by the basic block B . The longest path for a given processor resource r is the one that uses r the most. The $resMII(B)$ is given by the maximum longest path over all processor resources.

2.4 Cache/Memory Available Bandwidth

We define the *available memory bandwidth* of a basic block B , $M(B)$, as the number of additional memory references that can be issued in B without decreasing the initiation rate of operations in the loop. As with processor resources, we use a conservative estimate of available memory bandwidth. $M(B)$ is defined as the minimum available bandwidth over all control paths that execute B :

$$M(B) = \min_{p: B \in p} \{M(p)\}$$

To compute the available memory bandwidth of a control path p , $M(p)$, we estimate the number of potential cache misses, $m(p)$, in each path p of the loop. The available memory bandwidth of p is a function of $m(p)$ and the number of outstanding misses that the processor can have before it stalls, k . In our experiments we defined the available memory bandwidth of a control path p as the difference between the two: $M(p) = k - m(p)$.

2.5 Prefetch Profitability Condition

Given a basic block B that contains a memory reference L that is part of a pointer-chasing recurrence, the decision to prefetch for L is based on the prefetch profitability condition below:

Condition 1 (Prefetch Profitability) *Prefetching for a load L in a basic block B is considered profitable if the following condition holds:*

$$(resMII^P(L) \leq recMII^P(L)) \wedge (M(B) > 0)$$

where $resMII^P(L)$ and $recMII^P(L)$ are the resource and recurrence conservative lower bounds on initiation interval for the load L after the prefetch sequence is inserted in the code, and $M(B)$ is the available memory bandwidth of the basic block B before the prefetch sequence is inserted in the code.

Condition 1 states that prefetching for a load L in a basic block B is profitable only if the initiation rate of L is still limited by recurrences in the data dependence graph *and* the potential cache miss introduced by the prefetch will

¹ We do not consider the loop's back edge when applying the longest path algorithm to the CFG. As a consequence, our current implementation of the profitability analysis ignores the use of processor resources carried over iterations.

```

DoPREFETCH( $P, E, V$ )
1.  $C \leftarrow$  pointer-chasing recurrences;
2.  $R \leftarrow$  Prioritized list of induction pointer loads in  $C$ ;
3.  $N \leftarrow$  Prioritized list of field loads (not in  $C$ );
4.  $O \leftarrow R + N$ 
5. Mark each  $o$  in  $O$  as a cache miss;
6. for each  $L$  in  $O, L \in B$ 
7.   do if  $recMII^P(L) \geq resMII^P(B)$  and  $S(B) > 0$ 
8.     then add prefetch for  $L$  to  $B$ ;
9.     mark  $L$  as cache hit;
10.  endif
11. endfor

```

Fig. 3. Prefetch algorithm.

not block memory fetching from the cache. This is a conservative approach because, even when there is no available bandwidth between the cache and the main memory, prefetching at the correct address could improve performance by starting memory accesses earlier.

2.6 Prefetching Field Loads

Prefetching for an induction pointer load enables prefetching of [*field* loads. Field loads access memory through induction pointers but are not part of the recurrence cycles. Prefetching for the load of an structure field does not require additional address computations. Although prefetching of fields does not affect recurrences, it may still be beneficial to prefetch data for field loads because such prefetches will allow for the overlapping between memory accesses to fields and other computations performed by the loop. Prefetching of field loads is considered profitable if there are enough unused processor and memory resources for such prefetching in every affected control flow path. A field load prefetch is issued only if the field is not expected to lie in the same cache line as its induction pointer (see Section 3.1).

3 Prefetch Algorithm

The goal of the prefetch algorithm is to introduce prefetching whenever the extra usage of resources does not reduce the initiation rate of operations in the loop beyond the constraint imposed by loop recurrences.

We prioritize the loads in pointer-chasing recurrences (step 2-4 in Figure 3) according to the frequency of execution of the basic blocks to which they belong. In step 5 we mark the loads in the list O as cache misses. Observe that once a

pointer-chasing recurrence is identified, it is reasonable to predict that without prefetching the loads of such pointers will result in cold cache misses.²

For each load operation L in the priority list, we compute the available memory bandwidth of basic block B to which L belongs, $M(B)$. If $M(B)$ is greater than zero, it means that a new cache miss caused by the prefetch instruction will not cause the processor to stall. Therefore there is unused memory bandwidth available for a prefetch instruction. In this case we insert the prefetch operations in B and compute $recMII^P(L)$ and $resMII^P(L)$. If $recMII^P(L) \geq resMII^P(L)$, whenever the loop executes L , its initiation rate is still constrained by the recurrences and not by resource usage. In this case we mark L as a cache hit for future memory bandwidth availability estimates, otherwise we remove prefetch operations from B .

Inserting prefetch operations for a load L in a basic block B may change the resource usage and the memory bandwidth availability for all basic blocks that share a control path with B . It also changes the $recMII$ of recurrences to which L belongs. Therefore these quantities must be recomputed for each operation that we consider for prefetching.

3.1 Computation of the Memory Bandwidth Availability

The available memory bandwidth of a basic block B of a loop, $M(B)$, is a function of the maximum number of cache misses that may occur on any of the control paths that execute B . Observe that the available memory bandwidth of all operations that lie on the same control path changes when a prefetch sequence is inserted in the path. We estimate the number of cache misses that can occur in one iteration of the loop based on the coloring of the cache miss interference graph.

Definition 1. *The **miss interference graph** is an undirected graph $G(V, E)$ formed by a set of vertices, V , representing the memory operations in the loop, and a set of edges, E , representing the **interference** relationship between the memory operations. Two memory operations interfere if they may both cause a cache miss in the same loop iteration.*

When the cache miss interference graph is colored, memory operations that interfere with each other are assigned distinct colors. Thus, the minimum number of colors necessary to color the miss interference graph corresponds to the maximum number of misses that may occur in one iteration of the loop, regardless of the control path executed by that iteration.

We use the following set of heuristic interference rules to build the miss interference graph:

- memory references that are relative to a global pointer do not interfere with any other memory references (we assume that they hit in the cache);

² Some exceptions to this rule include inner loops traversing short linked lists, or circular pointer chasing structures. Our framework does not attempt to identify such situations.

- memory references that are relative to the stack pointer do not interfere with any other memory references (we assume that they hit in the cache);
- memory references where the memory address is loop invariant do not interfere with any other memory references;
- two memory references at address **base + constant offset** do not interfere if the bases are the same and the difference in their offsets is less than K , where K is a parameter in our algorithm (we used $K = 32$, i.e. the cache line size).
- two memory references that are executed in mutually exclusive control paths do not interfere with each other.

The available memory bandwidth for a basic block B , is $M(B) = k - N$, where N is the number of distinct colors necessary to color the set of loads in B and all its adjacent nodes in the cache miss interference graph, and k is the number of outstanding cache misses that the processor can support before it stalls.

4 Performance Evaluation

In order to evaluate our prefetch technique, we have collected 10 programs that spend a significant portion of its execution time executing loops that traverse linked lists. Our test suite includes two SPEC CINT95 benchmarks (126.gcc and 130.li), seven SPEC CINT2000 (181.mcf, 197.parser, 300.twolf, 175.vpr, 253.perlbmk, and 254.gap) and two applications (mlp is a perceptron simulator and ft is a minimum spanning tree algorithm). For the benchmarks in the SPEC benchmark suite we used the reference input. Table 1 shows the execution time of each program using three prefetch strategies: no prefetching, prefetching without profitability analysis (all induction pointers in all pointer chasing loops are prefetched), and induction pointer prefetching with the profitability analysis.

Preliminary evaluation results show that (1) identification of the induction pointers is a good basis for prefetching; (2) the prefetching technique presented in this paper achieves good speedups for programs that spend significant time in pointer chasing loops; and that (3) balancing the loop recurrences and the processor resource usage is necessary in order for prefetching to be effective.

4.1 Experimental Framework

We implemented our prefetch framework in the SGI MIPSpro Compiler suite. This suite consists of highly-optimizing compilers for Fortran, C, and C++ on MIPS processors. It implements a broad range of optimizations, including inter-procedural analysis and optimization, loop nest rearrangement and parallelization, SSA-based global optimization, software pipelining, global and local scheduling, and global register allocation [3,10,17].

We performed our experiments on an SGI Onyx machine with a 195 MHz MIPS R10000 processor, 32 KB, two-way associative, non-blocking, primary

Table 1. Execution time in seconds with and without prefetching.

Benchmark	Execution Time			Performance Improvement	
	No Prefetch	No Prof. Analysis	With Prof. Analysis	No Prof. Analysis	With Prof. Analysis
181.mcf	3,396 sec.	2,854 sec.	2,699 sec.	16.0 %	20.5 %
ft	517 sec.	436 sec.	333 sec.	15.6 %	35.5 %
mlp	632 sec.	580 sec.	538 sec.	8.3 %	14.9 %
175.vpr	1,771 sec.	1,765 sec.	1,761 sec.	0.3 %	0.6 %
130.twolf	2,540 sec.	2,657 sec.	2,531 sec.	-4.6 %	0.4 %
254.gap	1,174 sec.	1,226 sec.	1,207 sec.	-4.4 %	-2.8 %
130.li	285 sec.	293 sec.	292 sec.	-2.8 %	-2.5 %
253.perlbmk	2,062 sec.	2,131 sec.	2,104 sec.	-3.3 %	-2.0 %
197.parser	2,180 sec.	2,245 sec.	2,189 sec.	-3.0 %	-0.5 %
126.gcc	122 sec.	123 sec.	122 sec.	-0.7 %	-0.1 %

cache and 1MB secondary cache. We measured wall-clock time for each benchmark under the IRIX 6.5 operating system with the machine running in a single user mode. The results reported are an average of three runs, and there was no noticeable difference on the measurements obtained in each of the three separate runs of the benchmarks.

Implementing our prefetch technique for an out-of-order processor such as the MIPS R10000 processor has advantages and disadvantages. The out-of-order scheduler of the MIPS R10000 dynamically pipelines loops that cannot be pipelined by existing compilers. Pipelining these loops is essential in order for the improved iteration rate with prefetching to be reflected in the performance of the code. On the other hand, the out-of-order issuing of instructions makes the measurement of performance difficult for two reasons: first, the dynamic scheduler may move prefetches closer to their target loads reducing prefetch efficiency; second, an out-of-order processor can reorder memory accesses, and therefore affect the cache miss behavior.

Finally, the prefetch distance in our experiments has been set to prefetch 2 iteration in advance. We found that this distance is the most effective for improving the L1 cache performance. Prefetching 1 iteration in advance, on the other hand, reduces performance penalty when prefetching is counterproductive.

4.2 When Pointer Prefetching Works

The prefetching algorithm presented in this paper achieved significant (over 15%) improvement in performance of the mcf, ft, and mlp (with a remarkable 35% for ft) on top of all the standard optimizations of an industry-strength compiler.

We noticed that a very small number of loops are responsible for most of the performance gain in those programs. For example, in the mcf, a single loop accounts for 40% of the program execution time. This loop has two induction pointers and both of them access memory locations at regular intervals that

exceed the cache line size. The iteration count of this loop is comparatively large, up to 400 iterations, and the control path taken most often is constrained by the pointer chasing recurrence. Prefetching is very effective in hiding this loop's primary cache miss latencies and has a significant impact on its performance. *ft* spends about 80% of its time in a loop that traverses a linked list of vertices of a graph. No other computation is done in that loop and the loop's initiation rate is constrained only by the pointer-chasing recurrence. The *mlp* program has two linked list loops where the majority of the execution time is spent. One of these loops updates the weights of the synapses of neurons in a multi-layer perceptron, while the other updates the gradient used to back-propagate the output error. The initiation rate of these loops is constrained by the pointer chasing recurrences. The trip count is moderate, 25 iterations on average, but large enough that prefetching makes a significant difference.

4.3 When Pointer Prefetching Does Not Help

On the other hand, speculative induction pointer prefetching is not as effective in a number of programs. The two main reasons for this are: (1) short loop trip counts, and (2) irregular memory access pattern caused by the control flow in the loop. In such cases, our profitability analysis has been effective in keeping the negative impact of prefetching reasonably small.

For example, although the *gcc* extensively uses linked lists, prefetching does not have significant impact because those lists are short and rarely suffer primary cache misses. *li* spends much of its time in a number of pointer chasing loops that operate on trees. Tree traversal using loops rather than recursive function calls resembles the traversal of linked lists. Tree traversal does not have stride regularity though, and speculative prefetching is counterproductive (we measured a 2.5% performance degradation and 20% increase in the number of cache misses when prefetching is added). Another program, *parser*, has a number of linked-list loops. However, these lists are either hash table lists (randomly placed in memory), or their location in memory is randomized by the memory allocator. Thus prefetching is ineffective and results in moderately higher L1 cache misses and TLB misses. Nonetheless, with profitability analysis the performance degradation for *parser* is only 0.5%.

These results suggest an improvement to our method of identification of memory references with potentially regular stride: complex control flow in a loop is a good indication that the stride may be irregular and prefetching must be avoided. This is subject for future work. On the other hand, use of profiling information would also be helpful in identifying loops with irregular strides or short trip counts.

5 Related Work

Compiler prefetching for array-based numeric applications takes advantage of the data locality and of the compiler ability to analyze numerical loops [12].

Application of prefetching to dynamically allocated irregular data structures is more difficult. Such data structures can not be efficiently analyzed by the compiler with respect to locality. Therefore in order for prefetching to be effective it is often necessary to speculatively predict addresses to drive prefetching. Many prefetching techniques have been proposed that try to address these problems.

Mowry and Luk use profiling information to identify potential cache misses [13]. Ozawa *et al.* develop a compiler analysis combined with instruction scheduling based on the observation that certain kinds of memory references that represent a small fraction of static instructions tend to be responsible for the majority of cache misses [14]. Finally, Lipasti *et al.* use the fact that often the data at an address passed as a parameter to a function call suffers a cache miss to issue prefetch for such cases [8].

Address speculation has typically been used in hardware prefetching schemes. As in this work, they exploit the fact that often the addresses referenced by loads and stores follow an arithmetic progression. By keeping track of the last effective address and of the address stride, previously unseen addresses are speculatively predicted [16, 7]. In particular, Selvidge [18] and Mehrotra [11] noticed the regularity in memory streams generated by linked list accesses. A similar hardware approach is to reproduce the address generation process in hardware and to perform it in advance of other computations [16].

Our approach is different from the one proposed by Luk and Mowry both in the identification of pointer chasing loops and in the implementation of prefetching [9]. They use high level declarations to identify Recursive Data Structures (RDS) and points-to analysis to detect when a pointer to a record is assigned a value that was obtained from the dereference of a pointer to the same record. The implementation of such induction pointer identification requires high-level information from the compiler front-end, and expensive pointer analysis for identifying a recursive data structure traversal. Their approach does not allow the application of prefetching exclusively to linked list traversals. In contrast our technique to identify induction pointers detects recurrence cycles associated with pointer-chasing. The complexity of our induction pointer detection is $O(n^2)$ in the number of nodes in the DDG of a loop, and thus is not nearly as complex as a points-to analysis. Notice that for the case of regular strides, Luk and Mowry's data linearization scheme requires that the data be re-mapped to contiguous memory location in order to use pointer arithmetic to compute the addresses. In our framework no data re-mapping is required.

Chilimbi, Hill, and Larus propose the use of cache-conscious data structures to improve the caching of pointer-based data structures. They propose to improve locality by re-engineering the allocation of such structures [2].

6 Conclusion

Prefetching for pointer-based applications has been recognized as a difficult problem because the compiler does not know what to prefetch and when to do so. In this paper we present a compiler prefetch method that is applicable to linked

lists. We identify candidates for prefetching as data accessed through induction pointers and propose a profitability analysis which effectively determines when such prefetching will be beneficial. The prefetch technique presented in this paper is remarkably simple and quite effective for some benchmarks, resulting in gains in performance of 15-35% over their performance after standard compiler optimizations, while displaying minimal performance degradation for others. This simplicity — a small set of extra instruction for dynamic address calculation, no data remapping, and no hardware support — allowed for an effective implementation in an industry-strength compiler. Moreover, there was no noticeable change in the time required to compile the benchmarks that we tested when the prefetch analysis was executed. Thus, with our framework, a compiler optimization option can be implemented in existing compilers. Such a option would allow existing processor architectures to deliver significant speedup for programs that access data structures with regular stride.

References

1. T. F. Chen and J. L. Baer. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers*, 44:609 – 623, May 1995.
2. T. M. Chilimbi, M. D. Hill, and J. R. Larus. Making pointer-based data structures cache conscious. *Computer*, 33(12):67–74, December 2000.
3. F. Chow, S. Chan, R. Kennedy, S-M Liu, R. Lo, and Peng Tu. A new algorithm for partial redundancy elimination based on SSA form. In *International Conference on Programming Languages Design and Implementation*, pages 273 – 286, 1997.
4. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press; McGraw-Hill Book Company, Cambridge, Massachusetts; New York, New York, 1990.
5. J. C. Dehnert and R. A. Towle. Compiling for the cydra 5. *The Journal of Supercomputing*, 7:181–227, May 1993.
6. J. Fu and J. Patel. Stride directed prefetching in scalar processors. In *International Symposium on Microarchitecture*, pages 102 – 110, 1992.
7. J. Gonzales and A. Gonzales. Speculative execution via address prediction and data prefetching. In *International Conference on Supercomputing*, pages 196 – 203, 1997.
8. M. Lipasti, W. Schmidt, S. Kunkel, and R. Roediger. SPAID: Software prefetching in pointer- and call-intensive environments. In *International Symposium on Microarchitecture*, pages 231 – 236, 1995.
9. C. K. Luk and T. Mowry. Compiler based prefetching for recursive data structures. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222 – 233, 1996.
10. S. Mantripragada, S. Jain, and J. Dehnert. A new framework for integrated global local scheduling. In *Conference on Parallel Architectures and Compilation Techniques*, pages 167–174, Paris, France, October 1998.
11. S. Mehrotra. *Data Prefetch Mechanisms for Accelerating Symbolic and Numeric Computation*. PhD thesis, University of Illinois at Urbana-Champaign, 1996.
12. T. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching*. PhD thesis, Stanford University, 1994.

13. T. Mowry and C. K. Luk. Predicting data cache misses in non-numeric applications through correlation profiling. In *International Symposium on Microarchitecture*, pages 314 – 320, 1997.
14. T. Ozawa, Y. Kimura, and S. Nishizaki. Cache miss heuristics and preloading techniques for general-purpose programs. In *International Symposium on Microarchitecture*, pages 243 – 248, 1995.
15. B. Rau. Iterative modulo scheduling. Technical Report HPL-94-115, HP Laboratories, 1995.
16. A. Roth, A. Moshovos, and G. Sohi. Dependence based prefetching for linked data structures. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 115 – 126, 1998.
17. J. Ruttenberg, G. R. Gao, A. Stouchinin, and W. Lichtenstein. Software pipelining showdown: Optimal vs. heuristic methods in a production compiler. In *International Conference on Programming Languages Design and Implementation*, pages 1–11, Philadelphia, PA, May 1996.
18. C. Selvidge. *Compilation-Based Prefetching for Memory Latency Tolerance*. PhD thesis, MIT, 1992.
19. A. Stoutchinin, J. N. Amaral, G. R. Gao, J. Dehnert, and S. Jain. Automatic prefetching of induction pointers for software pipelining. Technical Report 37, November 1999.
20. R. Tarjan. Enumeration of the elementary circuits of a directed graph. *SIAM Journal on Computing*, 2(3):211 – 216, September 1973.

Constant-Time Root Scanning for Deterministic Garbage Collection

Fridtjof Siebert

Institut für Programmstrukturen und Datenorganisation (IPD)
Universität Karlsruhe
Am Fasanengarten 5
76128 Karlsruhe, Germany
`siebert@jamaica-systems.de`

Abstract. Root scanning is the task of identifying references to heap objects that are stored outside of the heap itself, in global and local variables and on the execution stack. Root scanning is particularly difficult within an incremental garbage collector that needs to be deterministic or give hard real-time guarantees. Here, a method that allows exact root scanning is presented. The basic idea is to ensure that copies of all root references exist on the heap whenever the garbage collector might become active. This approach reduces the root scanning phase of the garbage collection cycle to an efficient constant-time operation. A Java virtual machine and a static Java byte-code compiler that use this technique have been implemented and analysed using the SPECjvm98 benchmark suite. This Java implementation allows for deterministic memory management as needed in real-time systems that is difficult to achieve with traditional methods to perform root scanning.

1 Introduction

With the rising popularity of modern object-oriented programming languages like Java [1] garbage collection has finally been accepted as a means for memory management, even though it typically brings a high degree of indeterminism to the execution environment. Nevertheless, Java is more and more promoted as a development tool even for real-time critical systems that require deterministic execution [2, 3].

The indeterminism of garbage collection has two aspects: the garbage collector causes pauses that are hard to predict while the automatic detection of free memory makes predictions on the memory demand of an application difficult.

To avoid long unpredictable pauses caused by the collector, incremental or concurrent garbage collection techniques are employed [4, 5, 6, 7]. A garbage collection cycle can be performed in small increments while the main application executes. Each collection cycle starts with the root scanning phase, during which all references outside of the heap are detected and the referenced objects on the heap are marked. This includes references on the local stacks of all threads, in processor registers and in global variables outside of the heap. After the root scan-

ning phase, the collector continues by recursively marking objects referenced by objects that have already been marked.

The root scanning of the stack of one thread typically requires stopping the corresponding thread during the time the references are scanned. This can cause pause times that are too long for time-critical real-time applications that require short response times.

In addition to these pause times, there is often not enough information available on where references are located and which references are live. Conservative techniques can be employed in this case. Conservative root scanning treats all bit patterns that happen to represent legal object addresses on the heap as if they were actual pointers to heap objects. This usually works well since it is unlikely that a random integer or float value on the stack is a legal object address, but it makes the memory management completely unpredictable. Another difficulty in this approach are dangling references that reference objects on the heap, but that represent dead variables that are no more used by the application.

A deterministic garbage collector that can be used in safety-critical systems requires exact information on the roots. Additionally, an incremental garbage collector that is to be employed in hard real-time systems has to guarantee that the pause times for root scanning are bounded and very short.

2 Related Work

Little work has been published in the area of root scanning for real-time garbage collection, but several papers describe mechanisms for conservative and exact root scanning.

The application of conservative root scanning in a *mostly copying* compacting garbage collector has been presented by Barlett [8]. This collector refrains from moving an object and changing its addresses if the objects might be referenced from a conservatively scanned root reference.

Boehm presents a practical implementation of a conservative garbage collector for C using conservative root scanning [9] and describes methods that reduce the likelihood for pointer misidentification during conservative root scanning [10]. Heap addresses that are found to be referenced by misidentified pointers are *blacklisted*, the referenced memory is not used for allocation such that future references to these addresses found during conservative root scanning will not cause the retention of memory.

Goldberg [11] describes a method for tag-free collection based on an idea by Appel [12]. Appel's approach uses the return address stored in a function's activation frame to determine the function an activation frame belongs to. When this function is known, the types of the variables within the activation frame can be determined. Goldberg makes use of the fact that in a single-threaded environment, root scanning might only occur at calls or memory allocations. Additional information can be associated with each call point that describes a function's frame at this point, taking into account the life span of local variables. On some architectures, this can be implemented without a direct runtime cost. Goldberg also proposes an extension of his approach for multi-threaded environments: If

garbage collection is needed, all running threads should be stopped at the next allocation or procedure call.

Diwan et. al. [13] propose the use of tables generated by an optimizing compiler that describe live pointers and values derived from pointers and that allow the garbage collector not only to find references but as well to update the references to compact the heap. For multi-threaded environments, an approach similar to Goldberg's is suggested: when GC is triggered, suspended threads are resumed such that they reach their next *GC-point*. The compiler ensures that the time needed to reach the next *GC-point* is bounded.

Agesen et. al. [14] compare exact root scanning with the conservative approach that is typically used in Java implementations. The average reduction in heap size they achieved when using exact root scanning is 11% for a suite of 19 benchmarks they analysed, while the effect was more dramatic on a few tests.

Stichnoth et. al [15] show that it is even feasible to avoid the need for *GC-points* and instead provide exact information on live root references for every machine instruction generated by a compiler. This avoids the need to resume suspended threads when garbage collection is triggered. *GC maps* are needed to hold the exact information, the space required for these maps is about 20% of the size of the generated code.

The disadvantage of all these approaches is that they do not provide means to avoid the pause time due to root scanning. Dubé, Feeley and Serrano propose to put a tight limit on the size of the root set such that the root scanning time is limited [7]. Since this will not be practical for all systems, they propose to scan roots incrementally, which would require the use of a *write barrier* for all root references and impose a high runtime cost.

Christopher presented an interesting approach that is based on reference counting [16] and that avoids the need of scanning root references that are stored outside of the heap [17]. The idea is that all objects with a reference count that is higher than the number of references from other heap objects to this object must be reachable from a root reference outside the heap. All the objects reachable from such a root reference are then used as the initially marked set in the marking phase of the garbage collector. The disadvantage is the high runtime overhead that is required to keep the reference counts accurate: any assignment between local reference variables needs to ensure correct adjustment of the reference counts.

3 The Garbage Collector

The technique for root scanning presented here is largely independent of the actual garbage collection algorithm. The explanation is therefore limited to the description of the mechanisms relevant for root scanning. We have implemented an incremental mark and sweep collector, but the technique might as well be applied to different incremental garbage collection techniques that have been presented in earlier publications [4, 5, 6, 7]. For compacting or copying techniques, additional difficulties might arise when updating of root references is required. The use of handles would be a solution here.

4 Synchronization Points

An important prerequisite for the root scanning technique presented in the next section is to limit thread switches and garbage collection activity to certain points during the execution of the application. The idea has been presented earlier [18] and similar mechanisms have been employed earlier [19]. When synchronization points are used, thread switching at arbitrary points during the execution is prohibited. Instead, thread scheduling can occur only at *synchronization points* that are automatically inserted in the code by the compiler or virtual machine. The implementation has to guarantee to insert the code required at a *synchronization point* frequently enough to ensure short thread pre-emption delays. Since thread switches are restricted to *synchronization points*, root scanning might also only occur at these points. A different thread might run and cause garbage collection activity only if all other threads are stopped at *synchronization points*.

A possible implementation of a *synchronization point* is shown in **Listing 1**. A global semaphore is used to ensure that only one thread is running at any time. This semaphore is released and reacquired to allow a different thread to become active. The code is executed conditionally to avoid the overhead whenever a thread switch is not needed. The thread scheduler has to set the global flag *synchronization_required* whenever a thread switch is needed.

```
...
if (synchronization_required == true) {
    ...
    /* allow thread switch */
    V(global_semaphore);
    P(global_semaphore);
    ...
}
...
```

Listing 1. Code for conditional *synchronization point*.

The use of *synchronization points* has several important effects on the implementation:

1. The invariants required by an incremental garbage collector do not have to hold in between two *synchronization points*. It is sufficient to restore the invariant by the time the next *synchronization point* is reached. This gives freedom to optimizing compilers allowing them to modify the code in between *synchronization points*, e.g., the compiler might have *write barrier* code take part in instruction scheduling (*write barrier* code is code that has to be executed when references are modified on the heap such that an incremental garbage collector takes the modification into account).
2. Between two *synchronization points*, no locks are required to modify the memory graph or global data used for memory management. In the context of au-

automatic memory management this affects *write barrier* code and allocation of objects that can be performed without the need for locks on the accessed global data structures since all other threads are halted at *synchronization points* and are guaranteed not to modify this data at the same time.

3. Exact reference information is required only at *synchronization points* since root scanning can only take place here.

5 Constant Time Root Scanning

The idea presented here is to ensure that all root references that exist have to be present on the heap as well whenever the garbage collector might become active. This means that the compiler has to generate additional code to store references that are used locally on the program stack or in processor registers to a separate *root array* on the heap. Each thread in such a system has its own private *root array* on the heap for this purpose.

All references that have a life span during which garbage collection might become active need to be copied to the *root array*. Additionally, whenever such a reference that has been stored is not used anymore, the copy on the heap has to be removed to ensure that the referenced object can be reclaimed when it becomes garbage. The compiler allocates a slot in the current thread's *root array* for each reference that needs to be copied to the heap. The *root array* might be seen as a separate stack for references.

To ensure that the garbage collector is able to find all root references that have been copied to the *root arrays*, it is sufficient to have a single *global root pointer* that refers to a list of all *root arrays*.

The root scanning phase at the beginning of a garbage collection cycle can be reduced to marking a single object: the object referenced by the *global root pointer*. Since all *root arrays* and all references stored in the *root arrays* are reachable from this *global root pointer*, the garbage collector will eventually traverse all *root arrays* and all the objects reachable from the root variables that have been copied to these arrays. Since all live references have been stored in the *root arrays*, all local references will be found by the garbage collector.

The effect of this approach is that the *root scanning* phase becomes part of the garbage collector's mark phase: While the collector incrementally traverses the objects on the heap to find all reachable memory it incrementally traverses all root references that have been stored in the *root arrays*.

To maintain the incremental garbage collector's invariant, it is important to use the required *write barrier* code when local references are stored into *root arrays*.

Another minor problem are root references in global (static) variables. A simple solution is to store all static variables on the heap in a way that they are also reachable from the *global root pointer*. If this is not possible, a possible solution could be to always keep two copies of all static variables, one copy at the original location and another one in a structure on the heap that is reachable from the *global root reference*.

6 Saving References in Root Arrays

Saving local references in *root arrays* on the heap is performance critical for the implementation: Operations on processor registers and local variables in the stack frame are very frequent and typically cheap. Storing these references in the *root arrays* and executing the *write barrier* typically requires several memory accesses and conditional branches. To achieve good performance the number of references that are saved to the heap must be as small as possible.

The garbage collector might become active at any *synchronization point*. From the perspective of a single method, the collector might as well become active at any call, since the called method might contain a *synchronization point*. It is therefore necessary to save all local references whose life span contains a *synchronization point* or a call point. For simplicity of terms, *synchronization points* and call points will both be referred to as *GC-points* in the following text.

It is not clear when the best time to save a reference would be. There are two obvious possibilities:

1. *Late saving*: All references that remain live after a *GC-point* are saved directly before the *GC-point*. The entry of the *root array* that was used to save the reference will then be cleared right after the *GC-point*.

2. *Early saving*: Any reference with a life span that stretches over one or several *GC-points* is saved at its definition. The saved reference is cleared at the end of the life span, after the last use of the reference. Note that a life span might have several definitions and several ends, so code to save or clear the reference will have to be inserted at all definitions and ends, respectively.

It is not obvious which of these two strategies will cause less overhead. *Early saving* might cause too many references to be saved, since the *GC-points* within the life span might never be reached, e.g., if they are executed within a conditional statement. *Late saving* might avoid this problem, but it might save and release the same reference unnecessarily often if its life span contains several *GC-points* that are actually reached during execution.

A third possibility analysed here is a mixture between *early* and *late saving*, which can be done as follows:

3. *Mixed*: Since *synchronization points* that use a conditional statement like the one shown in **Listing 1** are executed only when a thread switch is actually needed, it makes sense to use *late saving* for life spans that only contain *synchronization points* but no call points. In this case, the saving of the reference in the *root array* before the thread switch and the clearing of the entry in the *root array* when execution of the current thread resumes can be done conditionally within the *synchronization point*, as shown in **Listing 2**. Whenever a life span contains call points, *early saving* is used since it is likely that one of the call points is actually executed and requires saving of the variable.

7 The Jamaica Virtual Machine

Jamaica is a new implementation of a Java virtual machine and a static Java compiler that provides deterministic hard real-time garbage collection. Root scanning

```

...
ref := ...;
...
if (synchronization_required == true) {
    Save_To_Root_Array(ref);

    /* allow thread switch */
    V(global_semaphore);
    P(global_semaphore);

    Clear_From_Root_Array(ref);
}
...
use(ref);
...

```

Listing 2. Conditionally saving a reference *ref* at a synchronization point

is done as just described: The compiler generates additional code to save live references that survive *GC-points* into *root arrays* on the heap. There is only a single *global root pointer*, and the garbage collector's root scanning phase is reduced to marking the single object that is referenced by the *global root pointer*.

Other aspects of the implementation's garbage collector have been described in more detail in earlier publications [20, 21]: An object layout that uses fixed size blocks is used to avoid fragmentation. The garbage collection algorithm itself does not know about Java objects, it works on single fixed size blocks. It is a simple Dijkstra et. al. style incremental mark and sweep collector [4]. A reference-bit-vector is used to indicate for each word on the heap if it is a reference, and a colour-vector with one word per block is used to hold the marking information. These vectors exist in parallel to the array of fixed size blocks.

The garbage collector is activated whenever an allocation is performed. The amount of garbage collection work is determined dynamically as a function of the amount of free memory in a way that sufficient garbage collection progress can be guaranteed while a worst-case execution time of an allocation can be determined for any application with limited memory requirements [22]. This approach requires means to measure allocation and garbage collection work. The use of fixed size blocks gives natural units here: the allocation of one block is a unit of allocation while the marking or sweeping of an allocated block are units of garbage collection work.

The static compiler for the Jamaica virtual machine is integrated in the Jamaica builder utility. This utility is capable of building a stand-alone application out of a set of Java class files and the Jamaica virtual machine. The builder optionally does smart linking and compilation of the Java application into C code. The compiler performs several optimizations similar to those described in [23]. The generated C code is then translated into machine code by a C compiler such as *gcc*.

8 Write Barrier in Jamaica

The purpose of a *write barrier* is to ensure that all reachable objects are found by the incremental garbage collector even though the memory graph is changed by the application while it is traversed by the collector. Three colours are typically used to represent the state of objects during a garbage collection cycle: *white*, *grey* and *black*. *White* objects have not been reached by the traversal yet. *Grey* objects are known to be reachable from root references, but the objects that are directly referenced by a *grey* objects might not have been marked yet, i.e., they might still be *white*. The incremental garbage collector scans *grey* objects and marks all *white* objects that are reachable from the *grey* object *grey* as well. The scanned object is then marked *black*.

The *write barrier* has to ensure the invariant that no *black* object refers to a *white* object (alternative invariants and *write barriers* have been presented by Pinenen [24]). One way to ensure the invariant is to mark a *white* object *grey* whenever a reference to the *white* object is stored in an object. The marking is typically done using a few mark-bits that represent the colour of an object. The *write barrier* code then sets these bits to the value that represents *grey* whenever a reference to a *white* object is written into a heap object. *Card marking* has been proposed [25] as an alternative for generational garbage collectors. The heap is divided into *cards* of size 2^k words, and every card has an associated bit that is set whenever a reference is stored in a cell of the associated *card*. This technique allows the use of very efficient *write barriers* [26].

Using mark bits is very efficient, but it has the disadvantage that finding a *grey* object might take time linear in the size of the heap (the mark bits of all objects need to be tested), and a complete garbage collection cycle might require time quadratic in the size of the heap. This is clearly not acceptable for deterministically efficient garbage collection. For the collection cycle to be guaranteed to finish in linear time, finding a *grey* object has to be a constant-time operation. A means to achieve this is to use a linear list of all *grey* objects. Marking a *white* object *grey* then involves adding the object to the list of *grey* objects.

Listing 3 illustrates the *write barrier* code that is needed by Jamaica to store a reference in an object. Nothing needs to be done if a *null* reference is to be stored. For non-*null* references, the colour of the referenced object needs to be checked.

```
...
if (ref != null) {
    Object **colour = adr_of_colour(ref);
    if ((*colour) == white) {
        (*colour) = greyList;
        greyList = ref;
    }
}
obj->f = ref;
...
```

Listing 3. Write barrier code required in Jamaica when storing a reference *ref* in the field *f* of object *obj*.

```

...
/* save ref in root array: */
if (ref != null) {
    Object **colour = adr_of_colour(ref);
    if ((*colour)==white) {
        (*colour) = greyList;
        greyList = ref;
    }
}
root_array[ref_index] = ref;
...
/* clear ref in root array: */
root_array[ref_index] = ref;
...

```

Listing 4. Required code to store an object in the root array and to clear the entry.

If it is *white*, the object needs to be marked *grey*, i.e., added to the list of grey objects. One word per object is reserved for the colour value. The colours *white* and *black* are encoded using special values that are invalid object addresses. Any other value represents *grey* objects, these objects form a linked list with the last element marked with a special value *last_grey* that is also no valid object address.

The *write barrier* code from **Listing 3** needs to be executed whenever a reference is saved in the *root array*. **Listing 4** illustrates the code required to save a reference in the *root array* and to release it later.

9 Analysis Using the SPECjvm98 Benchmark Suite

As we have seen above, the best time to save root references is not obvious. To find a good approach, the tests from the SPECjvm98 [27] benchmark suite have been analysed using *late saving*, *early saving* and *mixed*. Only one test from the benchmark suite, *_200_check*, is not included in the data since it is not intended for performance measurements but to check the correctness of the implementation (and Jamaica passes this test).

For execution, the test programs were compiled and smart linked using the Jamaica builder. They were then executed on a single processor (333 MHz UltraSPARC-III) SUN Ultra 5/10 machine with 256MB of RAM running SunOS 5.7.

In addition to the performance of Jamaica, the performance using SUN's JDK 1.1.8, 1.2 and 1.2.2 [28] and their just-in-time compilers has been measured as well. However, these values are given for informative reasons only. A direct comparison of the garbage collector implementation is not possible due to a number of fundamental differences in the implementations (deterministic real-time vs. non-deterministic garbage collection, static vs. just-in-time compilation, etc.).

9.1 When To Save References

First, the implementation was instrumented to count the number of references that

are saved during the execution of all seven tests using the three different strategies. The results are presented in **Fig. 1**.

Late saving of references requires the largest number of references to be saved. In one test, *_213_javac*, *late saving* even lead to intermediate C code that contained routines that were too large to be handled by the C-compiler on our system (*gcc 2.95.2*). The large code overhead makes this approach impractical for some applications.

Compared to the other strategies, *late saving* causes a significantly larger number of references to be saved for all but one test. Since multi-threading does not play an important role in most of the tests (the only exception being *_227_mtrt*), this indicates that many life spans that contain a *GC-point* cause several call points to be executed. Execution of several call points might also be the case if the only *GC-point* in a life span is a call point that lies within a loop, while the life span extends over this loop and the single call point is executed several times at runtime.

Early saving also causes a high number of references to be saved. Obviously, many references are saved that do not need to be saved and that are handled better by the *mixed* strategy. The reason for this are *synchronization points* that lie within the life span, but that only infrequently cause thread switches.

In all cases, the lowest number of references were saved using the *mixed saving* heuristic. Compared to *late* and *early saving*, the difference is often a dramatic reduction of a factor between two and four.

9.2 Runtime Performance

The runtime performance of the tests was analysed next. For these measurements, the tests were recompiled without the instrumentation that was used in the previous section to count the number of saved root references. The results of the performance measurements are shown in **Fig. 2**. For the analysis, the heap size was set to 32MB for all tests but *_201_compress*, since it required more memory to execute it was run with a heap of 64MB. The runtime shown is the real time, it was measured for three runs of each test. The values shown are the times from the fastest of these three runs.

In addition to the performance of Jamaica, the performance of SUN's JDK 1.1.8, 1.2 and 1.2.2 [28] and their just-in-time compilers has been measured as well.

The runtime performance decreases with the number of references saved, hence the runtime performance of the *mixed* strategy is best in all cases.

Compared to Sun's implementation, the performance of the *mixed* strategy is similar to that of JDK 1.1.8 or 1.2, while the performance of JDK 1.2.2. was improved significantly. One can expect that better optimization in the compiler implementation and direct generation of machine code will allow improvement of the performance of Jamaica as well.

9.3 Runtime Overhead of Root Saving

With the number of saved variables from **Fig. 1** and the measured runtime performance shown in **Fig. 2** we are able to estimate the runtime overhead of root scan-

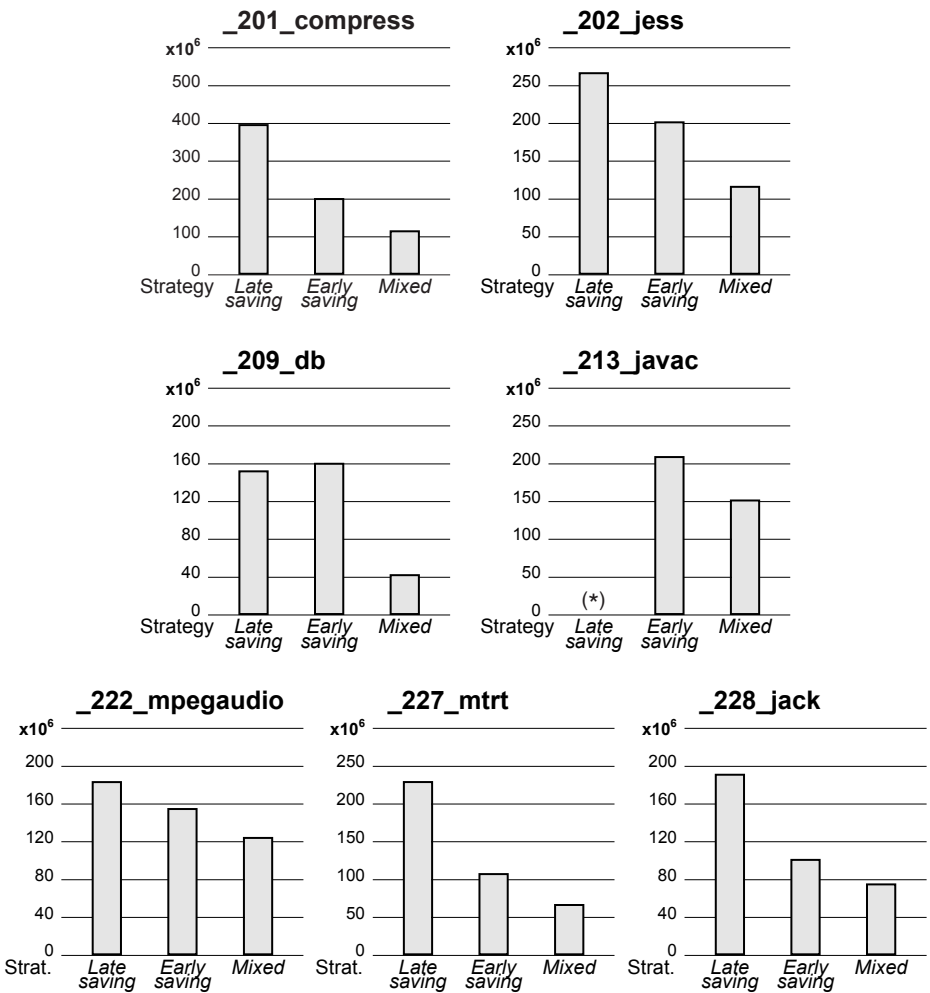


Fig. 1. Number of references saved in *root arrays* using *late saving*, *early saving* or *mixed* strategies.

(*) Late saving in this test caused some methods to become too large to be handled by the C-compiler (*gcc*)

ning. We assume that the cost to save the root references is linear in the number of references saved. This assumption holds if cache and instruction scheduling effects are ignored and the likelihood for a reference value to be *null* or the referenced object to be marked *white* is the same for the the three strategies *late*, *early* and *mixed saving*.

With these assumptions, linear regression analysis can be used to determine the execution time if no references needed to be saved. **Table 1** presents the results of this analysis. In addition to the estimated execution time with no referen-

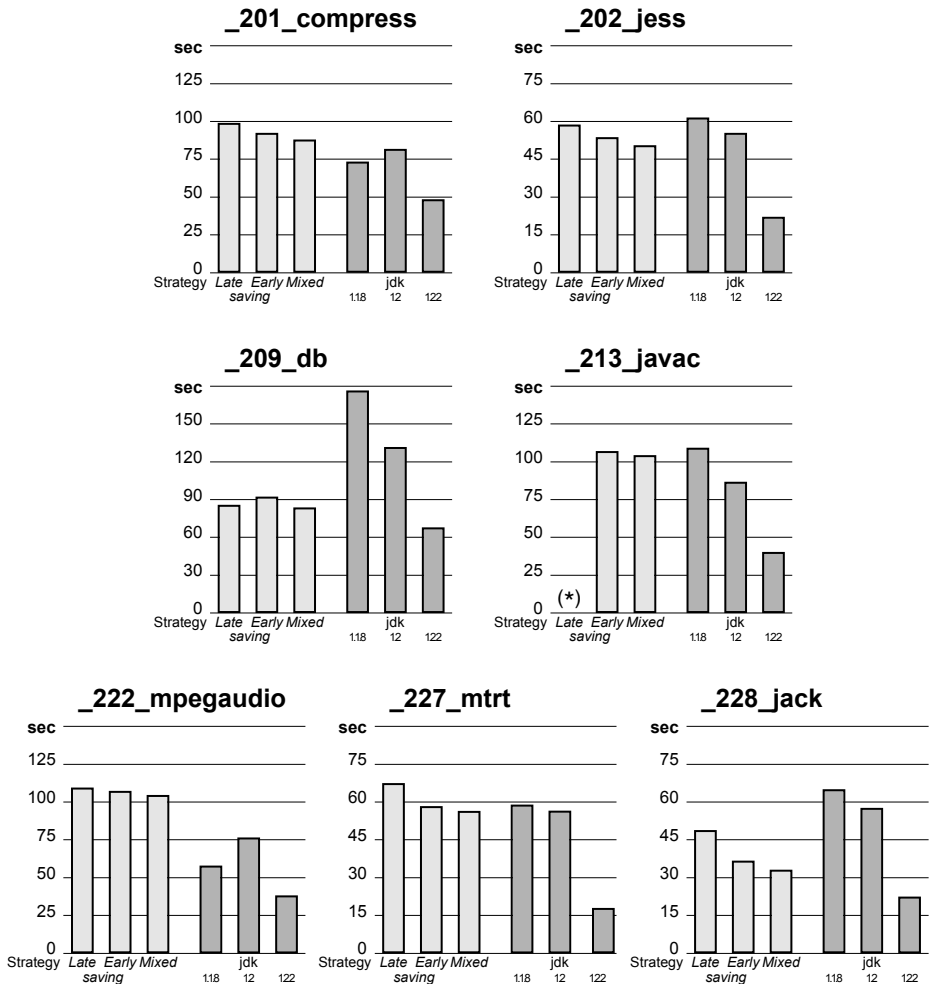


Fig. 2. Runtime performance of the SPECjvm98 benchmarks using Jamaica with *late saving*, *early saving* or *mixed* strategies and JDK 1.1.8, 1.2 and 1.2.2

(*) Late saving in this test caused some methods to become too large to be handled by the C-compiler (*gcc*)

ces saved, the percentage of overhead root saving imposes as a fraction of total runtime for all tests and the three strategies has been calculated.

The estimated overhead for root saving ranges from 3.3% up to 30.8% of the total execution time when the *mixed* strategy is used, with an average of 11.8%. The average overheads for *late* and *early saving* are significantly higher: 23.7% and 16.6%. There is only one test in which *late saving* has a lower overhead than *early saving*, *_209_db*. *Mixed* has the lowest overhead for all the tests.

The average overhead is fairly high, but one can expect that additional optimizations can help to reduce it further. Program-wide analysis can be employed

Benchmark	estimated runtime	root saving overhead		
	w/o root saving	late	early	mixed
_201_compress	79.7s	19.6%	13.9%	9.6%
_202_jess	43.9s	25.4%	18.5%	13.3%
_209_db	80.2s	6.6%	13.2%	3.3%
_213_javac	97.2s	-	9.2%	6.9%
_222_mpegaudio	94.4s	13.9%	12.1%	9.9%
_227_mtrt	51.3s	24.0%	12.1%	9.1%
_228_jack	23.0s	53.0%	37.6%	30.8%
Average		23.7%	16.6%	11.8%

Table 1. Extrapolated runtime without root saving overhead and percentage of runtime overhead for *late*, *early* and *mixed* saving strategies.

to avoid multiple saving of the same reference in the caller and the callee method. In addition to that, a more efficient implementation of the *write barrier* code that needs to be executed to save a reference will be possible if the implementation generates machine code directly. A global register could then be used to always hold the head of the grey list and another global register might be used to determine the colour entry associated with each object (the current implementation uses global variables that need to be read explicitly).

9.4 Memory Footprint

Another important impact of the root saving code is on the code size, which is an important factor for many applications in embedded systems with limited resources. The sizes of the executable binary files generated for the tests from the SPECjvm98 benchmark suite are presented in **Fig. 3**. The figures are the sizes of the 'strip'ped binary files compiled for SPARC/Solaris that were used for the performance measurements in **Fig. 2**.

The code size cost of *late saving* is very significant, the binary files are 35% to 86% larger compared to *early* and *mixed saving*. The smallest binary files are achieved using *early saving*, while the file sizes for *mixed* are between 0.2% and 1.7% larger.

10 Conclusion

A new technique to do exact root scanning in an incremental, deterministic garbage collected environment has been presented. The technique allows to avoid the pause times due to root scanning. The root scanning phase of the garbage collection cycle is reduced to a cheap constant-time operation.

Different strategies to implement the technique have been implemented in a Java environment and analysed. The performance was compared to current implementations that use non-deterministic garbage collection. It has been shown that the overhead of the technique is limited and allows performance comparable to current techniques. It permits a deterministic implementations that will allow new application domains for garbage collected languages like safety-critical controls with tight hard real-time deadlines.

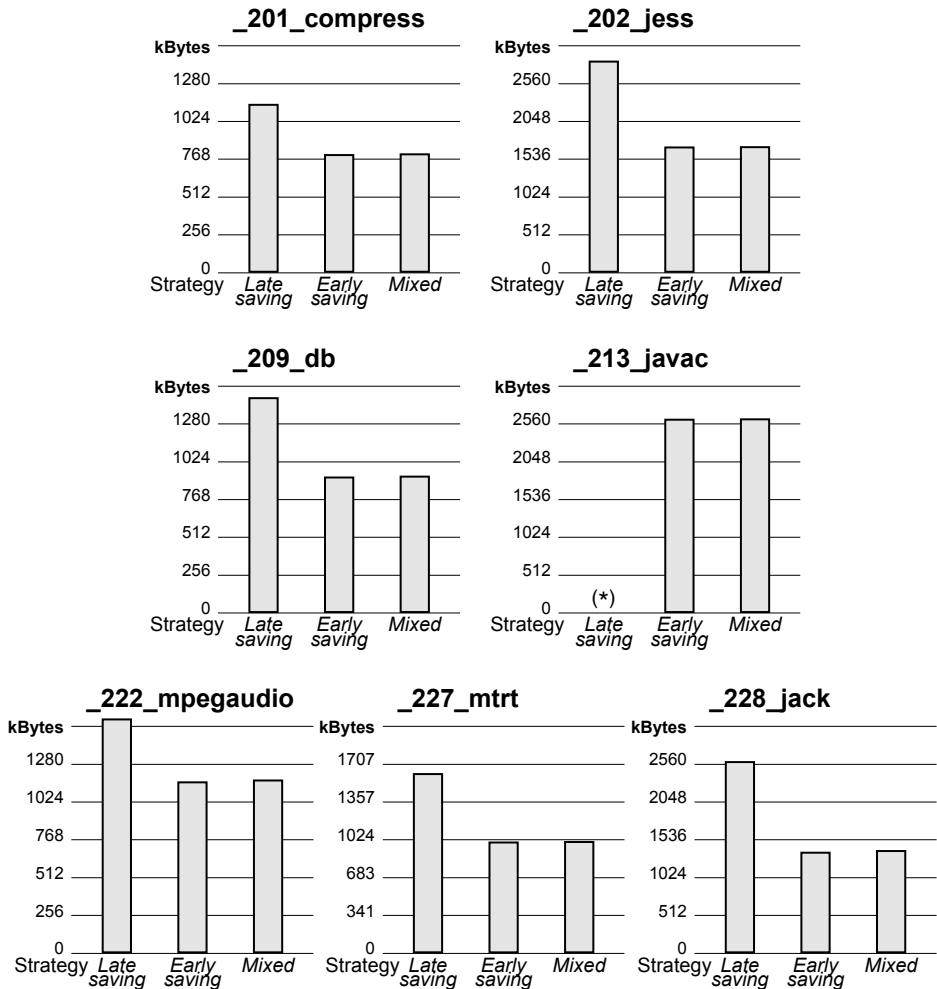


Fig. 3. File size of executable binary file for application using *late saving*, *early saving* or *mixed* strategies when compiled for SPARC/Solaris.

(*) Late saving in this test caused some methods to become too large to be handled by the C-compiler (*gcc*)

References

- [1] Ken Arnold and James Gosling: *The Java Programming Language*, 2nd edition, Addison Wesley, 1998
- [2] J-Consortium: *Real Time Core Extension for the Java Platform*, Draft International J-Consortium Specification, V1.0.14, September 2, 2000
- [3] The Real-Time Java Experts Group: *Real Time Specification for Java*, Addison-Wesley, 2000, <http://www.rtj.org>
- [4] Edsger W. Dijkstra, L. Lamport, A. Martin, C. Scholten and E. Steffens: *On-the-fly Garbage Collection: An Exercise in Cooperation*, Communications of the ACM,

- 21,11 (November 1978), p. 966-975
- [5] Henry G. Baker: *List processing in Real Time on a Serial Computer*. Communications of the ACM 21,4 (April 1978), p. 280-294, <ftp://ftp.netcom.com/pub/hb/hbaker/RealTimeGC.html>
 - [6] Damien Doligez and Georges Gonthier: *Portable, Unobtrusive Garbage Collection for Multiprocessor Systems*, POPL, 1994
 - [7] Danny Dubé, Marc Feeley and Manuel Serrano: *Un GC temps réel semi-compactant*, Journées Francophones des Langages Applicatifs, JFLA, Janvier 1996
 - [8] Joel F. Barlett: *Compacting Garbage Collection with Ambiguous Roots*, Digital Equipment Corporation, 1988
 - [9] Hans-Juergen Boehm: *Garbage Collection in an Uncooperative Environment*, Software Practice & Experience, Vol 18 (9), p. 807-820, September 1988
 - [10] Hans-Juergen Boehm: *Space Efficient Conservative Garbage Collection*, PLDI, 1993
 - [11] Benjamin Goldberg: *Tag-Free Garbage Collection for Strongly Typed Programming Languages*, PLDI, 1991
 - [12] A. W. Appel: *Runtime Tags Aren't Necessary*, Lisp and Symbolic Computation, 2, p. 153-162, 1989
 - [13] Amer Diwan, Eliot Moss and Richard Hudson: *Compiler Support for Garbage Collection in a Statically Typed Language*, PLDI, 1992
 - [14] Ole Agesen, David Detlefs, J. Eliot and B. Moss: *Garbage Collection and Local Variable Type-Precision and Liveness in Java™ Virtual Machines*, PLDI, 1998
 - [15] James Stichnoth, Guei-Yuan Lueh and Michal Cierniak: *Support for Garbage Collection at Every Instruction in A Java™ Compiler*, PLDI, 1999
 - [16] Donald E. Knuth: *The Art of Computer Programming*, Volume 1, 1973
 - [17] Thomas W. Christopher: *Reference Count Garbage Collection*, Software Practice & Experience, Vol 14(6), p. 503-507, June 1984
 - [18] Fridtjof Siebert: *Real-Time Garbage Collection in Multi-Threaded Systems on a Single Processor*, Real-Time Systems Symposium (RTSS'99), Phoenix, 1999
 - [19] Ole Agesen: *GC points in a Threaded Environment*, Sun Labs Tech report 70, 1998
 - [20] Fridtjof Siebert: *Hard Real-Time Garbage Collection in the Jamaica Virtual Machine*, Real-Time Computing Systems and Applications (RTCSA'99), Hong Kong, 1999
 - [21] Fridtjof Siebert: *Eliminating External Fragmentation in a Non-Moving Garbage Collector for Java*, Compilers, Architectures and Synthesis for Embedded Systems (CASES), San Jose, 2000
 - [22] Fridtjof Siebert: *Guaranteeing Non-Fisruptiveness and Teal-Time Deadlines in an Incremental Garbage Collector (corrected version)*, International Symposium on Memory Management (ISMM'98), Vancouver, 1998, corrected version available at <http://www.fridi.de>
 - [23] M. Weiss, F. de Ferrière, B. Delsart, C. Fabre, F. Hirsch, E. A. Johnson, V. Joloboff, F. Roy, F. Siebert, and X. Spengler: *TurboJ, a Bytecode-to-Native Compiler*, Languages, Compilers, and Tools for Embedded Systems (LCTES'98), Montreal, in Lecture Notes in Computer Science 1474, Springer, June 1998
 - [24] Pekka P. Pirinen: *Barrier techniques for incremental tracing*, International Symposium on Memory Management (ISMM'98), Vancouver, 1998
 - [25] Paul R. Wilson and Thomas G. Moher: *A card-marking scheme for controlling inter-generational references in generation-based GC on stock hardware*, SIGPLAN Notices 24 (5), p. 87-92, 1989
 - [26] Urs Hölzle: *A Fast Write Barrier for Generational Garbage Collectors*, OOPSLA, 1993
 - [27] *SPECjvm98 benchmarks suite, V1.03*, Standard Performance Evaluation Corporation, July 30, 1998
 - [28] *Java Development Kit 1.1.8, 1.2 and 1.2.2*, SUN Microsystems Inc., 1998-2000

Goal-Directed Value Profiling

Scott Watterson and Saumya Debray

Department of Computer Science
University of Arizona
Tucson, AZ 85721
{saw,debray}@cs.arizona.edu

Abstract. Compilers can exploit knowledge that a variable has a fixed known value at a program point for optimizations such as code specialization and constant folding. Recent work has shown that it is possible to take advantage of such optimizations, and thereby obtain significant performance improvements, even if a variable cannot be statically guaranteed to have a fixed constant value. To do this profitably, however, it is necessary to take into account information about the runtime distribution of values taken on by variables. This information can be obtained through value profiling. Unfortunately, existing approaches to value profiling incur high overheads, primarily because profiling is carried out without consideration for the way in which the resulting information will be used. In this paper, we describe an approach to reduce the cost of value profiling by making the value profiler aware of the utility of the value profiles being gathered. This allows our profiler to avoid wasting resources where the profile can be guaranteed to not be useful for optimization. This results in significant reductions in both the time and space requirements for value profiling. Our approach, implemented in the context of the *alto* link-time optimizer, is an order of magnitude faster, and uses about 5% of the space, of a straightforward implementation.

1 Introduction

Compilers can exploit knowledge that an expression in a program can be guaranteed to evaluate to some particular constant at compile time via the optimization known as constant folding [14]. This is an “all-or-nothing” transformation, however, in the sense that unless the compiler is able to guarantee that the expression under consideration evaluates to a compile-time constant, the transformation cannot be applied. In practice, it is often the case that an expression at a point in a program takes on a particular value “most of the time” [5]. As an example, in the SPEC-95 benchmark *perl*, the function *memmove* is called close to 24 million times: in almost every case, the argument giving the size of the memory region to be processed has the value 1; we can take advantage of this fact to direct such calls to an optimized version of the function that is significantly simpler and faster. As another example, in the SPEC-95 benchmark *li*, a very frequently called function, *livecar*, contains a *switch* statement where one of the case labels, corresponding to the type *LIST*, occurs over 80% of the time; knowledge of this fact allows the code to be restructured so that this common case can be tested separately first, and so does not have to go through the jump table, which is relatively expensive. As these examples suggest, if we know that certain values occur very frequently at

certain program points, we may be able to take advantage of this information to improve the performance of the program. This information is given by a *value profile*, which is a (partial) probability distribution on the values taken on by a variable when control reaches the program point under consideration at runtime. Our experience with value-profile-based optimizations indicate that they can produce significant speedups for non-trivial programs [15].

Unfortunately, existing approaches to obtaining value profiles tend to be very expensive, both in time and space. For example, in an implementation of “straightforward” value profiling by Calder *et al.*, executables instrumented for value profiling are more than 30 times slower, on the average, than the uninstrumented executables. The reason for this is that their profiling decisions are made without any knowledge of the potential utility of the profiles, i.e., the ways in which the profiles will be used for optimization. Our experiments indicate that of all the variables and program points that are candidates for value profiling in a program, typically only a tiny fraction actually yield value profiles that lead to profitable optimizations¹. This means that in value profilers based on existing techniques, much of the overhead incurred in the course of profiling represents wasted work.

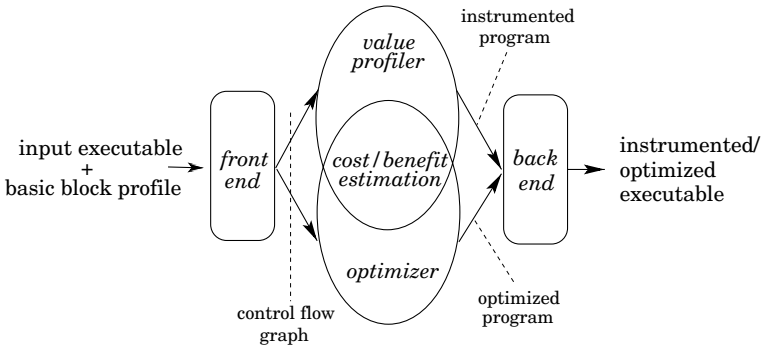


Fig. 1. Alto system structure

In this paper, we describe a system that avoids these problems by tightly integrating the value profiler with the optimizer that uses the results of value profiling (see Figure 1). This gives the value profiler a mechanism for estimating the utility of a value profile at a given program point, and allows it to identify profiling candidates—i.e., variables (memory locations, registers) at particular program points—that are clearly not worth profiling, i.e., that can be guaranteed to not yield optimization opportunities. This is done both during the instrumentation phase, when the profiler decides where to insert

¹ A significant reason for this is that, as discussed in Section 4, code optimized to exploit runtime value distributions must be guarded by a test, and the cost of this runtime test must be taken into account when weighing the profitability of the optimization.

instrumentation code for value profiling, and during the actual profiling itself, when the instrumentation code so inserted can determine whether it is worth continuing to profile a particular profiling candidate. By being able to use the cost-benefit considerations guiding the optimizer, the profiler is able to identify and prune value profiling candidates that have no possibility of contributing to any optimization. The resulting system is highly selective: typically, fewer than 1% of all possible profiling candidates are chosen for actual profiling. This selectivity leads to significant improvements in profiling performance, yielding order-of-magnitude improvements in both the space and time requirements for profiling, while retaining all of the optimization opportunities.

There has been a great deal of work on profiling techniques [23,13] and profile-guided code optimization [6,7,17]. However, to the best of our knowledge this is the first work that aims to make the profiler *goal-directed*, i.e., aware of the potential utility of the profiling information that it is gathering, and able to modify its profiling actions based on this awareness, in order to reduce profiling overheads.

2 System Overview

The work described here has been implemented in the context of *alto*, a link-time optimizer we have constructed for the Compaq Alpha architecture [16]. *Alto* can be used to rewrite executable files, either to instrument them for profiling, or to carry out code optimization. If profiling is carried out, a command line option can be used to control whether basic block profiles, edge profiles, or value profiles are gathered.

This paper focuses on value profiling. The gathering of value profiles is a two-stage process. We first use *alto* to instrument the input program—say, *a.out*—for basic block profiling. This yields an instrumented executable *a.bblprof.out* that is then executed using representative “training inputs” to obtain a basic block profile. Using this basic block profile, the original executable *a.out* is instrumented again, this time to gather value profiles. This yields a second instrumented executable *a.valprof.out*. Finally, *a.valprof.out* is executed with the training input to obtain a value profile.

For subsequent optimization, the basic block and value profiles so gathered are fed back to *alto*, which uses them to apply various code transformations on the input program *a.out*.

3 Mechanics of Value Profiling

Our value profiling technique is inspired by (and largely based upon) work by Calder *et al.* [5]. We keep track of the top *N* values occurring in a given register *r* at a given program point *p*. In addition to the top *N* values, we include a field that counts the number of values that are not recorded in the table; we label this the *other* field. Table 1 shows an example of a value profile, where *N* = 6. The field labelled *other* refers to values other than 1, 9, 20, 23, 100, or 0.

Each time execution reaches *p*, we profile the value contained in register *r*. If the value is in the table, we increment the count of that value. If the value contained in *r* is not already in the table, then we attempt to insert it. If there is no room in the table for the new value, we increment the count of the *other* field. In the above example table, if register

Table 1. Example Value Profile Table

<i>Value</i>	<i>Count</i>
1	2635
9	1093
20	1395
23	244
100	140
0	410
<i>other</i>	4698

r contains the value 100, the count of 100 in the table would simply be incremented. If r contained the value 2, then the count of *other* would be incremented, since there is no room in the table to insert 2.

The number of entries in the value profiling table has a significant impact on the running time of the value profiler. The table must be large enough that the most frequent entry will be in the table, but as the table grows, the profiler must look at more entries each time it reaches a profiling point. In this paper, we chose a value profiling table with 6 entries, plus the *other* field. This table size was the smallest size that captured most optimization opportunities. Further analysis of table sizes can be found in [10].

In some cases, the most frequently occurring value overall is not one of the first N distinct values. We would still like to capture this value during value profiling. We provide a mechanism for allowing later values into the table by periodically cleaning the lower half of the table. This means that the values are sorted based on their counts, and the $N/2$ least frequently occurring values are evicted (their counts are added to the *other* count). Though cleaning is a fairly heavy process, it happens infrequently. Our experiments indicate that, except for very small values of the cleaning interval, the actual cleaning frequency does not significantly affect either speed or quality of profiling. Given this insensitivity, we chose—in part to simplify comparison of results—the same cleaning interval as Calder *et al.* [5], namely, 1000. In other words, the table is cleaned after the execution has reached p 1000 times. After a table has been cleaned once, we must make sure that a new value can enter the steady part of the table before we clean again. We set the new value of the cleaning interval to $1000 +$ the count of the last entry in the steady part. This means that if a new value occurs almost exclusively between cleanings, its count should be higher than the last entry in the steady part of the table. It will then be moved into the steady part, evicting the least frequently encountered value.

Each program point selected for profiling incurs both an execution and a space cost. Profiling every register at every program point is obviously wasteful, so we must be more selective. Calder *et al.* instrumented every load instruction. We consider this the baseline approach, and discuss it in more detail in Section 5.1. Section 4 describes our approach.

4 Goal-Directed Value Profiling

To minimize the amount of wasted work during value profiling, we attempt to detect, as early as possible, those profiling candidates whose value profiles can be guaranteed to not yield useful optimizations. There are two possibilities for the detection of such unprofitable profiling candidates:

- (i) We may be able to tell, based on the basic block execution profile of the program, that specializing the program based on the values of a variable at a given program point will not yield an improvement in performance, regardless of the value profile for the variable at that program point. In this case, we can avoid profiling this candidate altogether. Section 4.1 describes a cost model for value-profile-based code specialization, and Section 4.3 discusses how this model can be used for a cost-benefit analysis that allows us to avoid profiling unprofitable candidates.
- (ii) Even if we cannot eliminate a profiling candidate ahead of time as discussed above, we may find, as profiling progresses, that the value profile for a particular profiling candidate is simply not “good enough” to yield any profitable optimization, regardless of the values that may be encountered for that candidate during the remainder of the computation. When this happens, we can discontinue profiling this candidate. A natural place to do this is at the point where a value profile table is cleaned, since cleaning requires us to examine the table in any case. The details of how we determine whether it is worth continuing to profile a particular candidate are discussed in Section 4.4.

Calder *et al.* note that instructions other than loads may be highly invariant [4]. For example, consider the following code:

```
r1 ← load 0(r18)
r2 ← and r1, 0xf
```

In this sequence, the `and` instruction may always produce the same result (if all of the loads have the same low 4 bits) despite the load instruction never loading the same value twice.

This discovery makes value profiling potentially much more powerful. The optimizer can consider many potentially useful program points other than load instructions for specialization. These may include function arguments (profiled at function entry), switch statement selectors, and other such values. Calder reports that full profiling for all load instructions results in a 32x slowdown. Since there are many program points, our cost-benefit analysis is of critical importance in reducing the amount of time and space used for profiling. We consider all registers at all program points candidates, and use the cost-benefit analysis as discussed in Sec 4.3 to choose those candidates that may yield useful optimizations.

Our system focuses on code specialization, since that optimization is implemented in `alto`. If we change the optimizations used in `alto`, for instance to reduce indirect function call overhead [18,19], we need not change the profiler. Since the optimizer shares the cost-benefit decisions with the profiler, the profiler would simply select different program points, based on the expected benefit.

4.1 A Cost Model for Value-Profile-Based Code Specialization

Our approach uses value profiles primarily for value-based specialization. By this we mean the elimination of computations whose result can be evaluated statically. Suppose we have a code fragment C that we wish to specialize for a particular value v of a register (or variable) r . Conceptually, value-profile-based specialization transforms the code to have the structure **if** ($r == v$) **then** $\langle C \rangle_{r=v}$ **else** C where $\langle C \rangle_{r=v}$ represents the residual code of C after it has been specialized to the value v of r . The test '**if** ($r == v$) ...' is needed because the value of r is not a compile-time constant, i.e., we cannot guarantee that r will not take on any value other than v at that program point.

Notice that, while the specialized code $\langle C \rangle_{r=v}$ may be more efficient than the original code C , the overall transformed code will actually be less efficient than the original code for values of r other than v because of the runtime test that has been introduced. There is thus a tradeoff associated with the transformation: if the optimized code is not executed sufficiently frequently, then the cost of performing the runtime test will outweigh the benefit of performing the value-based optimizations.

This optimization therefore requires a cost-benefit analysis to determine when to specialize a program. Our analysis attempts to assign a benefit value to each instruction. The benefit computation assigns a value to a $\langle \text{program point}, \text{register} \rangle$ pair. The benefit is an estimate of the savings from specializing code based on knowing the value of the register r at the program point p . This benefit computation has two components:

- (i) For each instruction I that uses the value of r available at p , there may be some benefit to knowing this value. The magnitude of this benefit will depend on the type of I , and is denoted by $\text{Savings}(I, r)$.
- (ii) It may happen that knowing the value of an operand register of an instruction allows us to determine the value computed by I . In this case, I is said to be *evaluable* given r . If I is evaluable given r , the benefit obtained from specializing other instructions that use the value computed by I are also credited to knowing the value of r at p . The indirect benefits so obtained from knowing the value of r in instruction I are denoted by $\text{IndirBenefit}(I, r)$.

The savings obtained from knowing the operand values for an individual instruction is essentially the latency of that instruction, if knowing the operand values allows us to determine the value computed by that instruction, and thereby eliminate that instruction entirely² (our implementation uses latency figures for various classes of operations based on data from the Alpha 21164 hardware reference manual):

$$\text{Savings}(I, r) = \text{if Evaluable}(I, r) \text{ then Latency}(I) \text{ else } 0.$$

Let $\text{Uses}(p, r)$ denote the set of all instructions that use the value of register r that is available at program point p . Then the benefit of knowing the value of a register r at program point p is given by the following:

² The benefit estimation can be improved to take into account the fact that for some instructions, knowing some of the operands of the instruction may allow us to strength-reduce the instruction to something cheaper even if its computation cannot be eliminated entirely. While our implementation uses such information in its benefit estimation, we don't pursue the details here due to space constraints.

$$\text{Benefit}(p, r) = \sum_{I \in \text{Uses}(p, r)} (\text{ExecutionFreq}(I) \times \text{Savings}(I, r) + \text{IndirBenefit}(I, r))$$

$$\text{IndirBenefit}(I, r) = \text{if Evaluable}(I, r) \text{ then } \text{Benefit}(p', \text{ResultReg}(I)) \text{ else } 0.$$

Here p' is the program point immediately after I , and $\text{ResultReg}(I)$ the register into which I computes its result.

The equations for computing benefits propagate information from the uses of a register to its definitions. These equations may be recursive in general, since there may be cycles in the use-definition chain. `alto` computes an approximation of the solution to the benefit equations. This estimate is used in `alto` to select program points for value-based optimizations.

We use a simple decision function for estimating when the net benefit due to specialization for a given value profile is high enough to justify specialization for a particular value v :

$$\text{Benefit}(p, r) \times \text{prob}(v) - \text{TestCost}(r, v) \times \text{ExecutionFreq}(p) \geq \phi, \quad (1)$$

where $\text{prob}(v)$ is the probability of occurrence of the value v , $\text{TestCost}(r, v)$ denotes the cost of testing whether a register r has a value v ,³ and ϕ is an empirically chosen threshold. This means that we will choose program point p and register r for optimization if the benefit is high enough. Note that the cost of testing for the value v is taken into account by subtracting it from the benefit.

This benefit computation is somewhat limited. It does not consider different possible values when computing the benefit for knowing the result of an instruction. In some cases (such as if the value is always zero), this means our computation may miss some opportunities to greatly simplify the code. Using particular values, however, results in very high overhead for the cost-benefit computation. Our computation also does not attempt to precisely model the cache behavior of the specialized code. We considered several alternative computations, but most were either impractical or too costly to compute. Our benefit computation, while imprecise in some respects can be computed quickly, and produces noticeable speedup on significant benchmarks (see Figure 6).

4.2 Expression Profiling

The idea of value profiling can be generalized to that of *expression profiling*, where we profile the distribution of values for an arbitrary expression, not just a variable or register, at a given program point. Examples include arithmetic expressions, such as “the difference between the contents of registers r_a and r_b ” and boolean expressions such as “the value of register r_a is different from that of register r_b ” In general, the expressions profiled may not even occur in the program, either at the source or executable level.

Expression profiles are not simply summaries of value profiles: e.g., given value profiles for registers r_a and r_b , we cannot in general reconstruct how often the boolean

³ The cost of the test varies, depending on the value of v . Usually, this test requires at least two instructions, a compare and a branch. If no registers are free, the test cost is higher. The cheapest value to test against is zero, since that requires only a conditional instruction.

expression $r_a == r_b$ holds. Expression profiles are important for two reasons. First, they conceptually generalize the notion of value profiles by allowing us to capture the distribution of relationships between different program entities. Second, an expression profile may have a skewed distribution, and therefore enable optimizations, even if the value profiles for the constituents of the expression profile are not very skewed: for example, a boolean expression $r_a \neq r_b$ may be true almost all of the time even if the values in r_a and r_b do not have a very skewed distribution.

The expressions that we choose to profile are determined by considerations of the optimizations that they might enable. Our implementation currently targets two optimizations: *loop unrolling* and *load avoidance*. More detailed discussion of expression profiling can be found in [15].

4.3 Static Elimination of Unprofitable Candidates

When determining whether to profile a particular profiling candidate, we consider whether the net benefit from any specialization for that candidate would be high enough in the best possible case. For this we assume that $\text{prob}(v) = 1$ (i.e., v is the only value encountered at runtime), and $\text{TestCost}(r, v)$ is the cheapest possible, i.e., we test against the value 0. If, despite these optimistic assumptions, the benefit of performing optimization is outweighed by the cost of performing the test, this variable is eliminated as a candidate for profiling. In other words, a program point register pair $\langle p, r \rangle$ is selected for profiling if and only if,

$$\text{Benefit}(p, r) \geq \text{TestCost}(r, 0) \times \text{ExecutionFreq}(p) + \phi, \quad (2)$$

Using this selection algorithm has the additional property that profiling will tend to be inserted in the less frequently executed portions of the code. Inserting tests in the middle of a frequently executed loop is unlikely to result in speedups, and using a cost benefit model accurately predicts that profiling such an instruction would be a waste of resources.

This elimination of candidates is conservative, in the sense that no matter what distribution of values is observed at runtime, the compiler will never choose this candidate for specialization. This means that profiling these instructions would be a waste of resources. This elimination of candidates is also very effective: As shown in Table 2, fewer than 1% of possible program points are selected for profiling. This results in a reduction in execution time of 74% on average (see Figure 2).

4.4 Dynamic Elimination of Unprofitable Candidates

Given the cost-benefit computation `alto` uses for selecting optimization opportunities, we can determine before value profiling how frequently the most frequent value must appear in order to perform specialization. This frequency can be used to turn off profiling when it becomes unprofitable (i.e. the compiler will not perform the optimization). The cutoff threshold is computed by taking equation 2 and solving for $\text{prob}(v)$.

$$\text{prob}(v) \geq \phi + \frac{\text{TestCost}(r, 0) \times \text{ExecutionFreq}(p)}{\text{Benefit}(p, r)}, \quad (3)$$

Table 2. Program Points Profiled Using Different Techniques

Program	No. of Program Points			
	Total	Baseline	Static	Dynamic
compress	16749	3302	71	17
gcc	271899	32910	6459	871
go	65328	14710	1273	262
jpeg	49650	11728	189	31
li	32221	5404	152	33
m88ksim	40867	7535	211	54
perl	82462	16500	430	160
vortex	113236	23499	242	36

This gives a lower bound on the probability the most frequent value must have in order for the optimizer to perform specialization. Since we know the execution frequency of p from our basic block profile, we can substitute that into the equation for $\text{prob}(v)$, yielding the following:

$$\frac{\text{count}(v)}{\text{ExecutionFreq}(p)} \geq \phi + \frac{\text{TestCost}(r, 0) \times \text{ExecutionFreq}(p)}{\text{Benefit}(p, r)},$$

If we solve this formula for $\text{count}(v)$, we can compute the minimum count that the most frequently occurring value must have for the optimizer to perform specialization. We then compute our threshold by subtracting this minimum count from the execution frequency of p . We know that if the count of the *other* field ever exceeds this threshold, then the optimizer can not select this value for specialization (since $\text{count}(v)$ can not be high enough⁴). This means, that if the *other* threshold is exceeded, we should stop profiling this point, since it will no longer be selected for optimization.

The cutoff threshold is thus computed as follows (ϕ is the same empirically chosen threshold as in equation [II](#)):

$$\text{Threshold} = (1 - \frac{\phi + \text{TestCost}(r, 0) \times \text{ExecutionFreq}(p)}{\text{Benefit}(p, r)}) \times \text{ExecutionFreq}(p).$$

Each profiling table has a boolean flag added, which determines when value profiling takes place. If the flag is true, normal value profiling takes place, if not, the value profiling is skipped, and execution is returned to the original code. This boolean flag is checked at every execution of the original profiling point.

The boolean flag is set to true before profiling begins. Each time the value profiling table is cleaned, the table is inspected to see if profiling should continue. The count of the *other* field is compared to the threshold cutoff value, and if the threshold is exceeded, the boolean flag is set to false. Thereafter, value profiling will not be performed at this program point.

⁴ Strictly speaking, profiling could stop when the sum of all table entries other than the most frequent value exceed the threshold. For reasons of efficiency and simplicity of implementation, we consider the more conservative criterion given above.

Use of the cutoff threshold as computed above results in an additional 25% reduction in execution time beyond static elimination (see Figure 2). Again, this threshold is a conservative choice, since the compiler would never choose this instruction for optimization after exceeding the threshold as computed above.

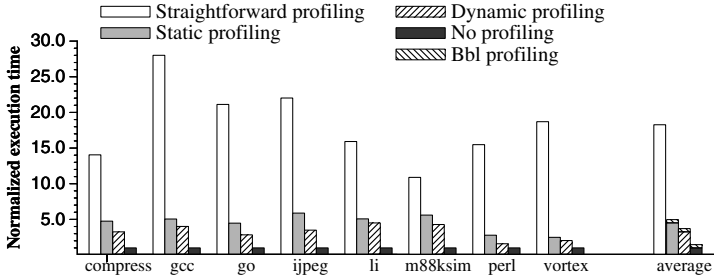


Fig. 2. Impact of Value Profiling on Execution Time (Relative to No Profiling)

5 Performance

We implemented our value profiling approach within the `alto` link-time optimizer. The programs used were the eight SPEC-95 integer benchmarks. The programs were compiled with the vendor supplied C compiler at maximum optimization (`cc -O4`) with additional flags to produce statically linked executables.

Our tests used the training input for all benchmarks. Times reported represent the average of ten runs, with the fastest and slowest times eliminated.

5.1 Baseline

Calder *et al.* instrument the register result of every load instruction [5]. To obtain a baseline for our results, we implemented this approach in `alto`. Table 2 shows the number of load instructions profiled using this approach. We implemented this straightforward approach only for comparison purposes.

In many cases such a straightforward approach will waste significant time and space. Most of the instructions being profiled will never be chosen by the optimizer for value-based optimizations. Profiling these points is a waste of resources, since the optimizer does not make use of the information obtained by the profiler. Knowing how the optimizer will make use of the profiles allows us to focus our profiler on those program points that may be chosen for value-based optimization.

Table 3. Space Requirements for Profiling

Program	Profiling Space (Bytes)		$Size_{Straightforward} / Size_{opt}$
	Straightforward	Optimized	
compress	343408	7384	0.0215
gcc	3422640	671736	0.1960
go	1529840	132392	0.0865
jpeg	1219712	19656	0.0161
li	583632	15808	0.0270
m88ksim	783640	21944	0.0280
perl	1716000	44720	0.0260
vortex	2443896	25168	0.0102
average	1231447	134115	0.0516

5.2 Experimental Results

Figure 2 shows the performance results, where No profiling represents the running time of the original executable. The Straightforward profiler profiled the result of every load instruction, as discussed in section 5.1. The Static profiler used static elimination of unprofitable candidates, as discussed in section 4.3. The Dynamic profiler added dynamic elimination of candidates as discussed in section 4.4.

Our baseline implementation profiled the result of every load instruction. This was quite costly, both in time and space, with a slowdown of 10-28 times the original code. After eliminating candidates unsuitable for profiling, the cost of profiling was reduced to 2.4-5x of the original execution. Space overhead was also significantly reduced, on average our approach uses 5% of the space of the straightforward approach. After adding the threshold optimization, the cost of value profiling was reduced to 1.5-4.5x of the original execution time. Table 3 shows the space overhead of value profiling.

The last column for Figure 2 shows the geometric mean of performance for each implementation. Above each bar we also show the runtime overhead of gathering basic block profiles, since we use this information to select value profiling opportunities. It can be seen that this additional overhead of basic block profiling is very small. This information would be gathered for optimization purposes no matter which implementation of value profiling is used.

5.3 Low Level Characteristics

We examined the performance of our profiling code using hardware performance counters. Unsurprisingly, we found that the Straightforward profiler used many more cycles than either the Static or Dynamic profilers. This is due to instrumenting many more program points than the other two approaches, which results in a much higher instruction count for the Straightforward profiler. Figure 3 shows the cycles for the original code, as well as the three profilers.

Adding the profiling cutoff threshold to the Static profiler resulted in a significant drop cycles and instructions. This is due to the Dynamic profiler abandoning program points

when they would no longer be selected by the optimizer. Figure 4 shows the number of loads executed, and Figure 5 shows the number of branch instructions executed.

For some benchmarks, such as compress, the profiling cutoff threshold was extremely effective. Many of the program points profiled were either extremely unpredictable (resulting in the cutoff being invoked quickly) or extremely predictable (resulting in the most frequent value being the initial table entry). For other benchmarks, the cutoff was less useful. This is largely due to the cleaning check only considering the *other* field when turning on or off profiling. If the value table has 3 values that each happen 33% of the time, then the cutoff will not be invoked. However, the flag is still checked every time the value profiling is invoked.

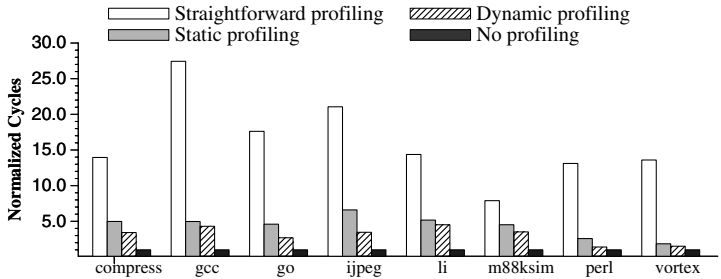


Fig. 3. Cycles used in benchmarks

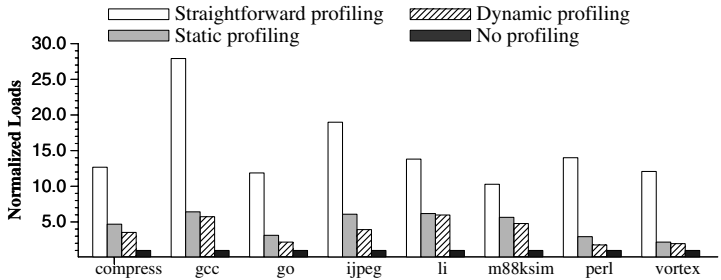


Fig. 4. Load Instructions Executed

5.4 Specialization Performance

This paper is focused on improving the performance of value profiling. Our system automates both the selection of profiling points for profiling and the selection of value profiles for specialization. Figure 6 shows the speedup of the specialized program over running

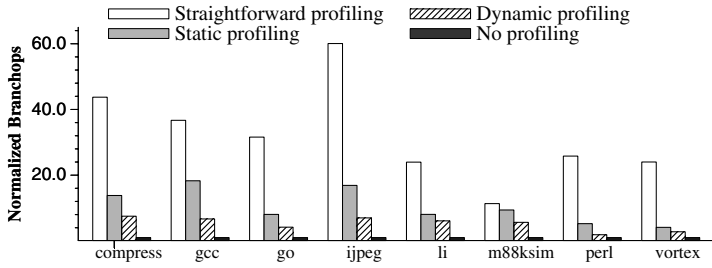


Fig. 5. Branch Instructions Executed

alto with all optimizations other than specialization. As Figure 6 shows, specialization yields speedups of up to 14.1% on significant programs such as the SPEC95 integer benchmarks. See [15] for further discussion of specialization performance in alto.

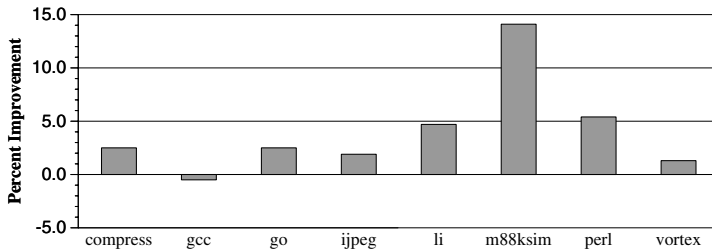


Fig. 6. Specialization Performance

6 Related Work

This work is closely related to the value profiling work of Calder *et al.* [5]. Their implementation supports profiling of both memory locations and registers. They report performance numbers for profiling the register result of every load instruction. Our current implementation profiles only registers. The mechanics used in our implementation are very similar to what Calder *et al.* refer to as “full” profiling.

Calder *et al.* also use convergent profiling to reduce the overhead of profiling in their system. Convergent profiling uses an invariance variable to determine if profiling is needed for a particular load instruction. This invariance variable is updated each time the load instruction is reached. If the table is not changing, profiling will be turned off for a while, and later turned back on for more profiling. An important invariant for us

was that the count of a value in the value profile was a lower bound on the number of times the value actually appeared at run time. At the same time, we would like the count to be as close to accurate as possible. The higher the frequency of a value, the more benefit that will accrue for that program point when it is considered for optimization. Convergent profiling loses information that “full” profiling retains. Calder *et al.* report an average slowdown of 10x for convergent profiling. We chose to use a cost-benefit analysis to reduce the number of program points profiled, rather than lose information through convergent profiling. Despite using “full” profiling, our overheads are considerably lower than those reported by Calder *et al.*

Also related is work on runtime code specialization, which uses a semi-invariant variable, along with its value to simplify the the code in the common case. Existing techniques require the programmer to annotate the code to identify candidates for code specialization [11,12]. Value profiling can then be used to determine which of the programmer-specified variables exhibit predictable behavior at runtime. Our implementation automates this process, identifying candidates for profiling, and then choosing among these candidates for optimization.

There is a large body of work relating to value prediction and speculation, e.g., see [11,12]. This involves using a combination of compiler generated information and runtime information. For example, a load instruction will often load the same value as it loaded the previous time it was executed. Predictable instructions such as these can be speculatively executed, and then checked at a later time to make sure that execution was correct. If the execution was incorrect, then recovery code must be executed.

Value profiling can help to classify predictable instructions for speculative execution. Value profiling could also indicate which of several predictors will best suit a particular instruction, increasing prediction accuracy.

7 Conclusion

Value-based optimizations are becoming increasingly important for modern compilers [9] [8]. Performing these optimizations relies on information gathered at runtime. Knowledge of how the optimizer will use this information can make the profiler much more efficient. This paper describes our techniques for reducing the cost of gathering value profiles, both in time and space, by tightly integrating the value profiler with the optimizer that uses the value profiles.

Our optimizer makes use of a careful cost-benefit analysis to choose optimization opportunities. We use this same analysis to select only those (program point, register) pairs for profiling that could be selected for optimization. This results in less than 1% of all program points being instrumented. Our profiler also makes use of this knowledge to modify its behavior while running, stopping profiling for program points that are no longer profitable. Our profiler will usually not instrument an instruction in the middle of a heavily executed loop, since the cost-benefit analysis can predict that such an instruction will not be chosen for optimization.

Our techniques result in a slowdown of 1.5-4.5x over the original code, down from a 10-28x slowdown for a straightforward approach. We also reduce the space requirements for a value profile to an average of 5% of that required by a straightforward approach.

References

1. J. Auslander, M. Philipose, C. Chambers, S. Eggers, and B. Bershad. Fast, effective dynamic compilation. In *SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 149–159, May 1996.
2. T. Ball and J. R. Larus, “Optimally Profiling and Tracing Programs”, *Proc. 19th. Symposium on Principles of Programming Languages*, Jan. 1992.
3. T. Ball and J. R. Larus, “Efficient Path Profiling”, *Proc. MICRO-29*, Dec. 1996.
4. B. Calder, P. Feller, and A. Eustace. Value profiling. In *30th Annual International Symposium on Microarchitecture*, pages 259–269, December 1997.
5. B. Calder, P. Feller, and A. Eustace. Value profiling and optimization. In *Journal of Instruction Level Parallelism*, 1999.
6. P. P. Chang, S. A. Mahlke, W. Y. Chen, and W. W. Hwu, “Profile-guided automatic inline expansion for C programs”, *Software Practice and Experience* vol. 22 no. 5, May 1992, pp. 349–369.
7. R. Cohn and P. G. Lowney, “Hot Cold Optimization of Large Windows/NT Applications”, *Proc. MICRO29*, Dec. 1996.
8. C. Consel and F. Noel. A general approach to run-time specialization and its application to c. In *23rd Annual ACM Symposium on Principles of Programming Languages*, pages 145–146, Jan 1996.
9. D. Engler, W. Hsieh, and M. Kaashoek. ‘c: A language for high-level, efficient, and machine-independent dynamic code generation. In *23rd Annual ACM Symposium on Principles of Programming Languages*, pages 131–144, Jan 1996.
10. P. Feller. Value profiling for instructions and memory locations. Master’s thesis, UCSD, 1998.
11. C. Fu, M. Jennings, S. Larin, and T. Conte. Software-only value speculation scheduling. Technical report, Department of Electrical and Computer Engineering, North Carolina State University, June 1998.
12. F. Gabbay and A. Mendelson. Can program profiling support value prediction? In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 270–280, Dec 1997.
13. A. J. Goldberg, “Reducing Overhead in Counter-Based Execution Profiling”, Technical Report CSL-TR-91-495, Computer Systems Lab., Stanford University, Oct. 1991.
14. S. S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufman, 1997.
15. Robert Muth, Scott Watterson, and Saumya Debray. Code specialization using value profiles. In *Static Analysis Symposium*, July 2000, pp. 340–359.
16. R. Muth, S. K. Debray, S. Watterson, and K. De Bosschere, “a1to: A Link-Time Optimizer for the Compaq Alpha”, *Software—Practice and Experience*, vol. 31 no. 1, Jan 2001, pp. 67–101.
17. K. Pettis and R. C. Hansen, “Profile-Guided Code Positioning”, *Proc. SIGPLAN '90 Conference on Programming Language Design and Implementation*, June 1990, pp. 16–27.
18. B. Calder and D. Grunwald, “Reducing Indirect Function Call Overhead in C++ Programs”, *Proc. 21st ACM Symposium on Principles of Programming Languages*, Jan. 1994, pp. 397–408.
19. U. Hölzle and D. Ungar, “Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback”, *Proc. SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI)*, June 1994, pp. 326–336.

A Framework for Optimizing Java Using Attributes

Patrice Pominville, Feng Qian, Raja Vallée-Rai, Laurie Hendren, and Clark Verbrugge

{patrice,fqian,kor,hendren}@cs.mcgill.ca
clarkv@ca.ibm.com

Sable Research Group, School of Computer Science, McGill University
IBM Toronto Lab

Abstract. This paper presents a framework for supporting the optimization of Java programs using attributes in Java class files. We show how class file attributes may be used to convey both optimization opportunities and profile information to a variety of Java virtual machines including ahead-of-time compilers and just-in-time compilers. We present our work in the context of Soot, a framework that supports the analysis and transformation of Java bytecode (class files) [21,25,26]. We demonstrate the framework with attributes for elimination of array bounds and null pointer checks, and we provide experimental results for the Kaffe just-in-time compiler, and IBM's High Performance Compiler for Java ahead-of-time compiler.

1 Introduction

Java is a portable, object-oriented language that is gaining widespread acceptance. The target language for Java compilers is Java bytecode which is a platform-independent, stack-based intermediate representation. The bytecode is stored in Java class files, and these files can be executed by Java virtual machines (JVMs) such as interpreters, just-in-time (JIT) compilers, or adaptive engines that may combine interpretation and compilation techniques, or they can be compiled to native code by ahead-of-time compilers. The widespread availability of JVMs means that Java class files (bytecode) have become a popular intermediate form, and there now exists a wide variety of compilers for other languages that generate Java class files as their output. One of the key challenges over the last few years has been the efficient execution/compilation of Java class files. Most of the work in this area has focused on providing better JVMs and the best performing JVMs now include relatively sophisticated static and dynamic optimization techniques that are performed on the fly, at runtime. However, another source of performance improvement is to optimize the class files before they are executed/compiled. This approach is attractive for the following reasons:

(1) Class files are the target for many compilers, and class files are portable across all JVMs and ahead-of-time compilers. Thus, by optimizing class files,

there is potential for a common optimizer that can give performance improvement over a wide variety of source language and target VM combinations.

(2) Class file optimization can be performed statically and only needs to be performed once. By performing the class file optimization statically we can potentially reduce the burden on JIT optimizers, and can allow for more expensive optimizations than can be reasonably performed at run-time. In general, one would want the combined effect of class file optimization and on-the-fly optimization.

Although optimizing class files is beneficial, there are limits to what can be expressed in bytecode instructions. Some bytecode instructions are relatively high-level, thus they hide details that may be optimizable at lower-level representations. For example, an access into an array is expressed as one bytecode instruction, but at run-time the array reference must be checked to ensure it is not null, the array bounds must be checked to ensure the index is in range, and appropriate exceptions must be raised if these checks fail. Clearly, one would like to avoid generating native code for the checks if a static analysis can guarantee that they are not needed.

We have developed a general mechanism for using class file attributes to encode optimization information that can be determined by static analysis of bytecode, but cannot be expressed directly in bytecode. The basic idea is that a static analysis of Java bytecode is used to determine some program property (such as the fact that an array index expression is in range), and this information is encoded using class file attributes. Any JVM/compiler that is aware of these attributes can use the information to produce better native code. In addition to array bound checks, such optimization attributes could be used for: register allocation [113], eliminating useless null pointer checks, stack allocation of non-escaping objects, devirtualization based on run-time conditions, specifying regions of potentially parallel code, or to indicate the expected behaviour of exceptions.

Attributes can also be used to convey profile information. Currently, advanced JVMs use on-the-fly profiling to detect hot methods, which may be optimized or recompiled on the fly. However, ahead-of-time compilers cannot necessarily make use of such dynamic information, and even for dynamic JVMs it may also be beneficial to use static information. For example, one could gather profile information from many executions, use information gathered from trace-based studies, or estimate profile information using static analysis. In these cases, the profile information could be conveyed via attributes.

In this paper we provide an overview of our general approach to supporting attributes in the Soot framework. We provide an infrastructure to support a very general notion of attributes that could be used for both optimization attributes and profile-based attributes. The paper is organized as follows. In Section 2, we briefly summarize the Soot framework, and outline our support for attributes. To demonstrate our approach we show how we applied it to the problem of eliminating array bounds checks, and we show how these attributes are expressed in our framework in Section 3. In order to take advantage of the optimization

attributes, the JVM/compiler processing the attributed class files must be aware of the attributes. We have modified both the Kaffe JIT and the IBM HPCJ (High Performance Compiler for Java) ahead-of-time compiler to take advantage of the array bound attributes, and we report experimental results for these two systems in Section 4. A discussion of related work is given in Section 5, and conclusions and future work are given in Section 6.

2 Attributes and Soot

Our work has been done in the context of the Soot optimizing framework [21,25,26]. Soot is a general and extensible framework to inspect, optimize and transform Java bytecode. It exposes a powerful API that lets users easily implement high-level program analyses and whole program transformations. At its core are three intermediate representations that enable one to perform code transformations at various abstraction levels, from stack code to typed three-address code. In Figure 1 we show the general overview of Soot. Any compiler can be used to generate the Java class files, and the Soot framework reads these as input and produces optimized class files as output. The Soot framework has been successfully used to implement many well known analyses and optimizations on Java bytecode such as common subexpression elimination, virtual method resolution and inlining [23]. All of these transformations can be performed statically and expressed directly in optimized Java bytecode. Until recently, the scope of these transformations was limited by the semantics and expressiveness of the bytecodes themselves. Hence, optimizations such as register allocation and array bounds check elimination could not be performed. The objective of the work in this paper was to extend the framework to support the embedding of custom, user-defined attributes in class files.

2.1 Class File Attributes

The de facto file format for Java bytecode is the class file format [17]. Built into this format is the notion of attributes that allows one to associate information with certain class file structures. Some attributes are defined as part of the Java Virtual Machine Specification and are essential to the correct interpretation of class files. In fact, all of a class's bytecode is contained in attributes. Attributes can also be user-defined and Java virtual machine implementations are required to silently ignore attributes they do not recognize.

The format of class file attributes is very simple and flexible: attributes consist of a name and arbitrary data. As shown in Figure 2, `attribute_name_index` is a 2 byte unsigned integer value corresponding to the index of the attribute's name in the class file's *Constant Pool*, `attribute_length` is a 4 byte unsigned integer specifying the length of the attribute's data and `info` is an array of `attribute_length` bytes that contains the actual uninterpreted raw attribute data. This simplistic model conveys great freedom and flexibility to those that wish to create custom attributes as they are unhindered by format constraints.

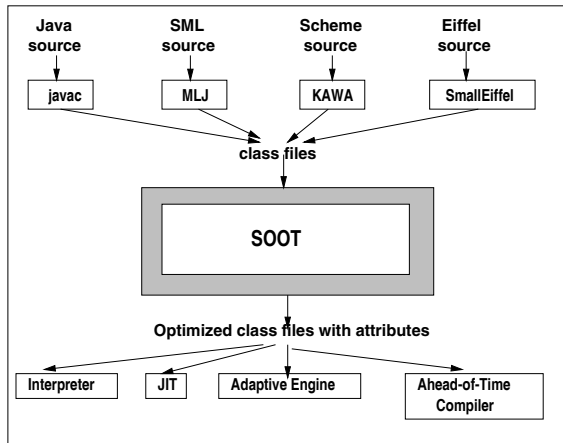


Fig. 1. General Overview

The only binding requirement is for a custom attribute's name not to clash with those of standard attributes defined by the Java Virtual Machine Specification.

```

attribute_info {
    u2 attribute_name_index;
    u4 attribute_length;
    u1 info[attribute_length];
}
  
```

Fig. 2. Class File Attribute Data Structure

Attributes can be associated with four different structures within a class file. In particular class files have one `class_info` structure as well as `method_info` and `field_info` structures for each of the class' methods and fields respectively. Each of these three structures contains an *attribute table* which can hold an arbitrary number of `attribute_info` structures. Each non-native, non-abstract method's attribute table contains a unique Code attribute to hold the method's bytecode. This Code attribute has an attribute table of its own, which can contain standard attributes used by debuggers and arbitrary custom attributes.

2.2 Adding Attributes to Soot

An Overview: Figure 3 provides a high-level view of the internal structure of Soot, with support for attributes. The first phase of Soot is used to convert the input class files into a typed three-address intermediate code called Jimple [6, 25]. In Jimple, each class is represented as a `SootClass`, and within each `SootClass`

there is a collection of `SootFields` and `SootMethods`. Each method has a method body which is represented as a collection of instructions, with each instruction represented as a `Unit`.

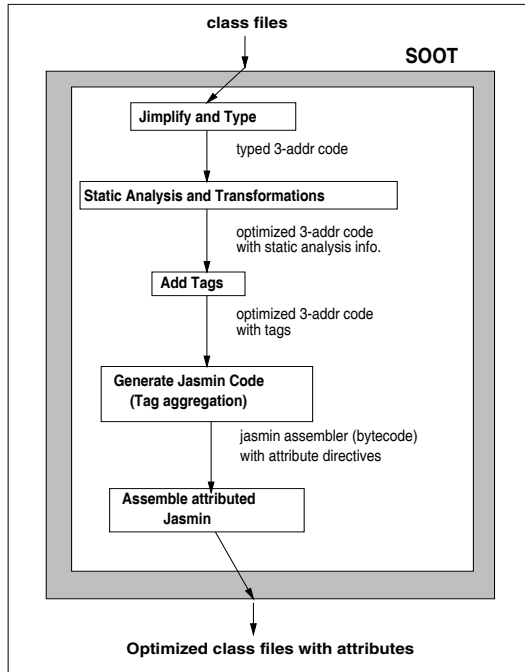


Fig. 3. Internal Structure of Soot

Jimple was designed to be a very convenient intermediate form for compiler analyses, and the second phase of Soot, as shown in Figure 3, is to analyze and transform the Jimple intermediate representation. There already exist many analyses in the Soot framework, but a compiler writer can also add new analyses to capture information that will be eventually output as class file attributes. Soot includes an infrastructure for intraprocedural flow-sensitive analyses, and implementing new analyses is quite straightforward. In Section 3 we discuss our example analysis for array bounds elimination.

After the analysis has been completed, analysis information has been computed, but one requires some method of transferring that information to attributes. In our approach this is done by attaching tags to the Jimple representation (third phase in Figure 3).

After tagging Jimple, the fourth phase of Soot automatically translates the tagged Jimple back to bytecode. During this phase the tags may be aggregated using an aggregation method specified by the compiler writer. Our system does

not directly produce class files, but rather it produces a form of assembly code used by the Jasmin bytecode assembler [11]. We have modified the Jasmin assembler language so that during this phase Jimple tags are converted to Jasmin attribute directives.

Finally, the fifth phase is a modified Jasmin assembler that can read the attribute directives and produce binary class files with attributes.

Hosts, Tags and Attributes: Attribute support in Soot has been achieved by adding two key interfaces: **Host** and **Tag**. Hosts are objects that can hold **Tags**; conversely, **Tags** are objects that can be attached to **Hosts**. These interfaces are listed in Figure 4. There are five Soot classes that implement the Host interface; these are **SootClass**, **SootField**, **SootMethod**, **Body** and **Unit**, the latter of which is Soot’s abstract notion of a bytecode instruction.

```
public interface Host {
    public List getTags(); /* gets list of tags associated with the host.*/
    public Tag getTag(String aName); /* gets a tag by name. */
    public void addTag(Tag t); /* adds a tag to the host. */
    public void removeTag(String name); /* removes a tag by name. */
    public boolean hasTag(String aName); /* checks if a tag exists.*/
}

public interface Tag {
    public String getName();
    public byte[] getValue();
}
```

Fig. 4. The Host and Tag Interfaces

Tags are meant to be a generic mechanism to associate name-value pairs to Host objects in Soot; they are not necessarily mapped into class file attributes. For this purpose, we have introduced the **Attribute** interface, which extends the **Tag** interface. Soot objects that are subtypes of **Attribute** are meant to be mapped into class file attributes; however, because the Soot framework uses the Jasmin tool to output bytecode, an **Attribute** object must actually be an instance of **JasminAttribute** for the translation to take place (see Section 2.2 for more information).

Compiler implementors can create application-specific subclasses of **JasminAttribute** and attach these to Hosts. There is a natural mapping between the aforementioned Soot classes that implement the Host interface and the attribute architecture present in class files as described in Section 2.1. **JasminAttributes** attached to a **SootClass** will be compiled into an entry in the attribute table of the corresponding class. **SootMethod** and **SootField** attributes are dealt

with similarly. Dealing with `JasminAttributes` attached to `Soot Units` is a bit trickier and is addressed in the following section.

Mapping Unit Attributes into a Method's Code Attribute Table: `Soot Attributes` attached to `Units` do not map trivially to a given class file structure as was the case for `SootClass`, `SootMethod` and `SootField` attributes, because `Units` naturally map to bytecode instructions, which do not have associated attribute tables. The obvious solution is to map all of a method's `Unit` attributes into entries in the method's `Code Attribute's` attribute table in the generated class file. Each entry will then contain the bytecode program counter (PC) of the specific instruction it indexes. This is what is done automatically by the Soot framework at code generation time. However, generating one `Code` attribute per `Unit` attribute can lead to undue class file bloat and increased processing and memory requirements by virtual machine's runtime attribute interpretation module. Often different instances of identical `Code Attribute` attributes should be expressed in a tabular format. For example instead of creating 10 null pointer check attributes for a method, it is more efficient to create a single redundant null pointer table as an attribute for these in the class file. The Soot framework allows an analysis implementor to easily create this table by providing the `TagAggregator` interface as outlined in Figure 5.

By implementing this interface and registering it in the class `CodeAttributeGenerator`, it is possible to selectively aggregate `Tags` attached to different `Unit` instances into a single `Tag`. A user can aggregate all `Attributes` generated by his/her analysis by iterating over a method's `Units` and calling the `aggregateTag` method on each of the `Tags` attached to a given `Unit`. The `produceAggregateTag` method is then called to produce a single aggregate attribute to be mapped into single attribute in the method's `Code Attribute's` attribute table.

```
public interface TagAggregator {
    public void aggregateTag(Tag t, Unit u);
    public Tag produceAggregateTag();
}
```

Fig. 5. The `TagAggregator` Interface

Extending Jasmin for Attribute Support : The Soot framework does not directly generate bytecode; instead it uses the Jasmin tool to do so. Jasmin specifies a textual assembler-like grammar for class files and transforms conforming input into binary class files. Because the Jasmin grammar does not provide constructs for expressing generic class file attributes, we have augmented it to accept and correctly process the added language constructs for attributes.

Informally, an attribute is encoded in Jasmin as a triple consisting of an attribute directive, the attribute's name and the attribute value in Base64.

The attribute directive is one of `.class_attribute`, `.method_attribute`, `.field_attribute` and `.code_attribute`. These directives must be produced in Jasmin code at specific locations:

- .class_attribute:** These must be found immediately *before* the class' field declarations.
- .field_attribute:** These must be found immediately *after* the field declaration they relate to.
- .method_attributes:** These must be found immediately *after* the method declaration they relate to.
- .code_attribute:** These must be found *before* the end of the method they relate to. Code attributes that correspond to instructions with specific bytecode PC values must express this symbolically. This is done by outputting a Jasmin assembler label before each bytecode that is indexed by some attribute. This label is then used as proxy for the PC of the bytecode it indexes in a Jasmin `.code_attribute` attribute. Labels are encoded inside an attribute's Base64 value data stream by surrounding the label name with the % symbol. When our modified Jasmin actually creates the class file attribute, it will replace the labels found in an attribute's data stream by the corresponding 16-bit bigendian PC value.

Figure 6(a) gives an example of a Java method and Figure 6(b) gives an extract of the attributed Jasmin assembler generated, showing the labeled bytecode instructions corresponding to the two array accesses in the program. For this example the generated class file attribute will be 6 bytes: 2 bytes for the PC represented by `label12`, followed by 1 byte for the Base64 encoded value `AA==` (no check needed), followed by 2 bytes for the PC represented by `label13` and finally 1 byte for the Base64 encoded value `Aw==` (array bounds checks needed).

<pre> public void sum(int[] a) { int total=0; int i=0; for (i=0; i<a.length; i++) total += a[i]; int c = a[i]; } </pre> <p>(a) Java source</p>	<pre> .method public sum([I)V ... label12: iaload ... label13: iaload return .code_attribute ArrayNullCheckAttribute "%label12%AA==%label13%Aw==" .end method </pre> <p>(b) attributed Jasmin</p>
---	---

Fig. 6. Attributed Jasmin

Summary: With all of these features the Soot framework is now well endowed with a simple-to-use, generic-attribute generation feature that is tightly integrated into its overall optimization support facilities. This enables analysis implementors to seamlessly augment their analysis with custom attribute support.

3 Attributes for Array Bounds Checks

In the previous section we outlined how we have integrated attributes into the Soot optimizing framework. In this section we illustrate the framework using an example of eliminating array bounds checks. We briefly describe the array bounds check problem, the analyses we use to find unneeded checks, and how to create bounds check tags and convert them into class file attributes by using the Soot framework. Finally, we show how to modify a JVM to take advantage of our attributes.

3.1 The Array Bounds Check Problem in Java

Java requires array reference range checks at runtime to guarantee a program's safe execution. If the array index exceeds the range, the runtime environment must throw an `IndexOutOfBoundsException` at the precise program point where the array reference occurs. For array-based computations, array bounds checks may cause a heavy runtime overhead, and thus it is beneficial to eliminate all checks which a static analysis can prove to be unneeded. In fact, several Java virtual machines implement array bounds check elimination algorithms in their JIT compilers [22,4]. In these systems the optimization is done at runtime as part of the translation of bytecode to native code. However, since the optimization is done at runtime, this approach has two limitations.

- (1) Only relatively simple algorithms can be applied because of time constraints.
- (2) They lack global information, such as field information and whole-program information. Usually a JIT compiler can not afford the expense of these analyses.

We have developed an algorithm that works at the bytecode level. By statically proving that some array references are safe and annotating these using class file attributes, an attribute-aware JIT can avoid generating instructions for array bounds checks without performing the analysis itself. The attributed class files can also be used by an ahead-of-time compiler, such as IBM's High Performance Compiler for Java.

Java requires two bounds checks, a lower bound check and upper bound check. The lower bound of an array is always a constant zero. The upper bound check compares the index with the array length. On popular architectures, such as IA-32 and PowerPC, both checks can be implemented by just doing an upper bound check with an unsigned compare instruction, since a negative integer is always larger than a positive one when it is interpreted as an unsigned integer.

Thus, in order to be really beneficial, one must eliminate both the upper and lower bound checks.

Another subtle point is that eliminating array bounds checks is often also related to eliminating null pointer accesses. Each array access, for example `x[i]`, must first check that the array `x` is not-null. In many modern compilers null pointer checks are performed by handling the associated hardware trap if a null pointer is dereferenced. In this case the machine architecture guarantees a hardware exception if any very low memory addresses are read or written. In order to do the upper array bounds check the length of the array must be accessed, and since the length of the array is usually stored at a small offset from `x`, this access will trap if `x` is null. Thus, the array bounds check gives a null pointer check for free. If the array bounds check is eliminated, then it may be necessary to insert an explicit null pointer check (since the address of `x[i]` may be sufficiently large to avoid the null pointer trap, even if `x` is null).

3.2 Static Analyses

We have developed two static analyses, *nullness analysis* and *array bounds check analysis*, using the Soot framework. Each of these analyses are implemented using the Jimple typed 3-address representation.

Nullness Analysis: Our nullness analysis is a fairly straightforward flow-sensitive intraprocedural analysis that is implemented as an extension of the `BranchedForwardFlowAnalysis` class that is part of the Soot API. The basic idea is that variable `x` is non-null after statements of the form `x = new()`; and statements that refer to `x.f` or `x[i]`. We also infer nullness information from condition checks of the form `if (x == null)`. Since the nullness analysis is intraprocedural, we make conservative assumptions about the effect of method calls.

Array Bounds Check Analysis: The core of the array bounds check algorithm is an intraprocedural analysis. For each method, it constructs an inequality constraint graph of local variables, integer constants, and other symbolic nodes (i.e. class fields, array references, and common subexpressions) similar in spirit to the work by Bodik et. al. [2]. The algorithm collects constraints of nodes and propagates them along the control flow paths until a fixed point is reached. For each array reference, the shortest distance from an array variable to the index indicates whether the upper bound check is safe or not, and the shortest distance from the index to the constant 0 determines the safety of lower bound check.

We have extended this basic analysis in two ways. The first handles the case where an array is assigned to a field in a class. Fields with `final` or `private` modifiers are analyzed first. Often these fields can be summarized by simply scanning all methods within the current class. Using these simple techniques we can determine whether a field is assigned a constant length array object that never changes and never escapes.

The second extension is used to find rectangular arrays. Multidimensional arrays in Java can be ragged (i.e. different rows in an array may have different lengths), and this makes it more difficult to get good array bounds analysis. However, in scientific programs arrays are most often rectangular. Thus, we have also developed a whole-program analysis using the call graph to identify rectangular arrays that are passed to methods as parameters. Rectangular arrays can have two different meanings, *shape rectangular* (each dimension has the same size, but subdimensions can be sparse in memory or aliased), or *memory-shape rectangular* (the object is created by 'multianewarray' bytecode and no subdimensions are ever assigned other array objects). The second type is stricter than the first. For bounds check analysis, shape rectangular is good enough.

3.3 From Analysis to Attributes

After the analysis phase the flow information is associated with Jimple statements. The next step is to propagate this information so that it will be embedded in the class file attributes. This is done by first tagging the Jimple statements, and then specifying a tag aggregator which packs all the tags for a method into one aggregated tag.

Format of Attribute: We first outline the attribute as it eventually appears in the generated class file. The structure of the array bounds attribute is quite straightforward. It has the name "ArrayNullCheckAttribute". Figure 7 shows the format of the array bounds check attribute as it will be generated for the class files.

```
array_null_check_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u3 attribute[attribute_length/3];
}
```

Fig. 7. Array Bounds Check Attribute

The value of `attribute_name_index` is an index into the class file's constant pool. The corresponding entry at that index is a `CONSTANT_Utf8` string representing the name "ArrayNullCheckAttribute". The value of `attribute_length` is the length of the attribute data, excluding the initial 6 bytes. The `attribute[]` field is the table that holds the array bound check information. The `attribute_length` is 3 times larger than the table size. Each entry consists of a PC (the first 2 bytes) and the attribute data (last 1 byte), totalling 3 bytes. These pairs are sorted in the table by ascending PC value.

The least 2 bits of the attribute data are used to flag the safety for the two array bounds checks. The bit is set to 1 if the check is needed. The null check

information is incorporated into the array bounds check attribute. The third lowest bit is used to represent the null check information. Other bits are unused and are set to zero. The array reference is non-null and the bounds checks are safe only when the value of the attribute is zero.

Add Attributes: It takes two steps to add attributes to class files when using the Soot annotation scheme. The attribute is represented as a **Tag** in the Soot framework. For the array bounds check problem we proceed as follows:

Step 1: Create an **ArrayCheckTag** class which implements the **Tag** interface. The new class has its own internal representation of the attribute data. In our implementation the **ArrayCheckTag** uses 1 byte to represent bounds checks as explained above. For each array reference, we create an **ArrayCheckTag** object. The tag is attached to a **Jimple** statement which acts as a **Host** for the array check tag.

Step 2: Create a class called **ArrayCheckTagAggregator** which implements the **TagAggregator** interface.

The aggregator will aggregate all array check tags for one method body. We then register the aggregator to the **CodeAttributeGenerator** class, and specify the aggregator as active. The aggregator generates a **CodeAttribute** tag when it is required to produce the aggregated tag. The **CodeAttribute** tag has the name "**ArrayNullCheckAttribute**", which is the attribute name in the class file.

Soot manages all **CodeAttribute** tags and produces a Jasmin file with the appropriate attribute directives, and finally Soot calls the extended Jasmin assembler to generate the class file with attributes.

3.4 Making a JVM Aware of Attributes

After generating the annotated class file, we need to make a JVM aware of attributes and have it use them to improve its generated native code. We modified both Kaffe's OpenVM 1.0.5 JIT and the IBM HPCJ ahead-of-time compiler to take advantage of the array bound attributes. Below we describe the modifications needed for Kaffe; the modifications to HPCJ are similar.

The KaffeVM JIT reads in class files, verifies them, and produces native code on demand. It uses the **_methods** structure to hold method information. We added a field to the **_methods** structure to hold the array bounds check attribute. Figure 8 shows the data structure.

When the VM reads in the array bounds check attribute of the **Code** attribute, it allocates memory for the attribute. The **<PC, data>** pairs are then stored in the attribute table. The pairs were already sorted by PC when written into the class file, so no sorting has to be done now.

The Kaffe JIT uses a large switch statement to generate native code for bytecodes. It goes through the bytecodes sequentially. We use the current PC as the key to look up the array bounds check attribute in the table before generating

<pre>typedef struct _methods { soot_attr attrTable; } methods;</pre>	<pre>typedef struct _soot_attr { u2 size; soot_attr_entry *entries; } soot_attr;</pre>	<pre>typedef struct _soot_attr_entry { u2 pc; u1 attribute; } soot_attr_entry;</pre>
---	--	--

Fig. 8. Modified Kaffe Internal Structure

code for array references. Because attribute pairs are sorted by ascending PC, and bytecodes are processed sequentially, we can use an index to keep the current entry in the attribute table and use it to find the next entry instead of searching the whole table. Figure 9 gives the pseudocode.

```
idx = 0;
...
case IALOAD:
    ...
    if (attr_table_size > 0) { /* attributes exist. */
        attr = entries[idx].attribute;
        idx++;
        if (attr & 0x03) /* generates bounds checks. */
            check_array_index(..);
        else
            if (attr & 0x04) /* generates null pointer check. */
                explicit_check_null(..);
    } else /* normal path */
        check_array_index(..);
```

Fig. 9. Using attributes in KaffeVM

Here, we turn off bounds check instructions when the array reference is non-null and both bounds are safe. We also insert null check instructions at the place where bounds check instructions can be removed but the null check is still needed.

4 Experimental Results

We measured four array-intensive benchmarks to show the effectiveness of the array bounds check attribute. The benchmarks are characterized by their array reference density, and the results of bounds check and null check analysis. These data are runtime measurements from the application programs, and do not include JDK libraries. We also measured the class file size increase due to attributes. Finally we report the performance improvement of benchmarks on

attribute-aware versions of KaffeVM and the IBM High Performance Compiler for Java.

4.1 Benchmarks

Table 1(1) shows benchmark characteristics¹. The third column describes the array type used in the benchmark: **s** represents single dimensional and **m** represents multidimensional². The last column shows array reference density of the benchmark, which is a count of how many array references per second occur in the benchmark. It is a rough estimate of the potential benefit of array bounds check elimination.

Table 1. Benchmarks

(1) Benchmarks				(2) Analysis results					(3) File size increase		
Name	Source	Type	Density	low	up	both	not null	all	Soot -W	with attr.	incr.
mpeg	specJVM98	s/m	19966466/s	89%	51%	50%	58%	26%	256349	276874	8.0%
FFT	scimark2	s	10262085/s	77%	61%	59%	97%	59%	2380	2556	7.4%
LU	scimark2	m	14718027/s	97%	64%	64%	68%	31%	1641	1907	16.2%
SOR	scimark2	m	13683052/s	99%	99%	99%	51%	50%	445	507	13.9%

4.2 Result of Analysis

Table 1(2) shows the percentage of checks that can be eliminated. This is measured by instrumenting the benchmark class file and getting dynamic numbers.

The first and second columns show the percentages of lower and upper bounds checks that could be safely removed. Usually we see that a much higher percentage of lower bounds can be proved safe than upper bounds. The third column shows the percentage of bounds checks where both upper and lower can be safely removed. The forth column shows the percentage of safe null pointer checks, and the fifth column shows the percentage of array references that are not null and with both bounds checks safe. Clearly we are most interested in the last column, when we determine that we can eliminate both bounds and we know the array is not null.

4.3 Class File Increase

The increase in file size due to adding attribute information is shown in Table 1(3).

¹ Note that we use the abbreviation **mpeg** for the benchmark **mpegaudio**.

² Multidimensional arrays are harder to analyze due to their more complex aliasing, as well as the possibility of being non-rectangular.

The first column shows the file size in bytes after Soot whole program optimization. The second column shows file size of the optimized class file including array bounds check attributes, and the relative increase in file size is listed in the last column. The file size increase depends primarily on the static count of array references—each array reference needs 3 bytes of attribute information, and the class itself needs a constant pool entry to store the attribute name. Note that in this prototype work we have made no effort to reduce or compress this information; significant improvements should be possible.

4.4 Kaffe Performance Improvement

We measured the KaffeVM (ver 1.0.5 with JIT3 engine) modified to take advantage of array bounds check attributes. It runs on a dual Pentium II 400MHz PC with 384Mb memory, Linux OS kernel 2.2.8, and glibc-2.1.3. The modified JIT compiler generates code for an array reference depending on the attribute. If no attribute is present, it generates bounds check instructions as in Figure 10(a). If the attribute shows safe bounds check and unsafe null check, it inserts null check code in place of the bounds check (Figure 10(b)). If both bounds checks and null check are safe, no instructions are added.

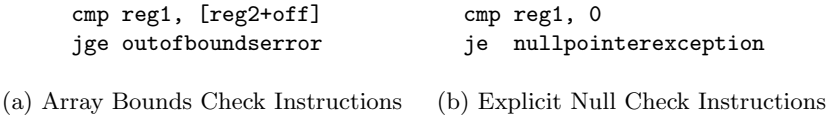


Fig. 10. Check Instructions

Table 2 gives the benchmark results for the attribute-aware KaffeVM. The “nocheck” column shows the running time without any bounds or null checks for the application classes, while the “with attr” and “normal” columns show the execution times of each benchmark with and without attributes respectively. Each benchmark gets some improvement roughly scaled according to the percentage of safe checks. Note that LU without checks actually has a performance degradation; this anomaly is discussed in the next section.

Table 2. KaffeVM Runtime

name	normal	nocheck	with attr
mpeg	80.83s	62.83s(22.3%)	72.57s(10.2%)
FFT	51.44s	48.84s(5.1%)	50.01s(2.8%)
LU	81.10s	81.88s(-0.9%)	78.15s(3.6%)
SOR	46.46s	41.23s(11.3%)	43.19s(7.0%)

4.5 High Performance Compiler Performance Improvement

The High Performance Compiler for Java runs on a Pentium III 500MHz PC with the Windows NT operating system. The structure of the High Performance Compiler is such that safe checks could be removed prior to its internal optimization phase, allowing subsequent optimizations to take advantage of the reduced code; this has resulted in a speed increase that does not correlate as well with the relative number of checks removed. Tables 3 and 4 show the benchmark times with and without internal optimizations—the last two columns in each table give the performance improvement when either just array bounds checks or just null pointer checks are completely removed; note that as with KaffeVM, there are some unexpected performance decreases.

Table 3. IBM High Performance Compiler without Optimizations

name	normal	nocheck	with attr	noarray	nonnull
mpeg	50.88s	29.96s(41.1%)	39.14s(23.1%)	30.64s(39.8%)	47.94s(5.8%)
FFT	28.22s	25.09s(11.1%)	26.59s(5.8%)	25.15s(10.9%)	28.85s(-2.2%)
LU	39.99s	28.83s(27.9%)	32.33s(19.2%)	28.92s(27.7%)	38.39s(4.0%)
SOR	24.16s	15.46s(36.0%)	15.55s(35.6%)	15.18s(37.2%)	23.96s(0.8%)

Table 4. IBM High Performance Compiler with Optimizations On

name	normal	nocheck	with attr	noarray	nonnull
mpeg	21.27s	15.93s(25.1%)	20.33s(4.4%)	17.12s(19.5%)	20.82s(2.1%)
FFT	17.39s	15.34s(11.8%)	19.45s(-11.8%)	16.08s(7.5%)	18.58s(-6.8%)
LU	21.50s	14.84s(30.8%)	21.27s(1.1%)	15.03s(30.1%)	21.49s(0.0%)
SOR	11.93s	8.88s(25.6%)	8.88s(25.6%)	8.88s(25.6%)	11.92s(0.1%)

Generally with the High Performance Compiler results, we see a very slight improvement in the running time due to null pointer check elimination (“normal” column and “nonnull” column respectively), and a significantly larger improvement due to array bounds check elimination (“noarray” column). This reflects the relative cost of the two operations—where possible the High Performance Compiler implements null pointer checks by handling the associated hardware trap if a null pointer is dereferenced. The machine architecture guarantees a hardware exception if any very low memory addresses (e.g. zero) are read or written. Thus, since most null pointer checks are required because of an impending dereference or write anyway, the null pointer check can be implemented as an implicit byproduct of the subsequent code (see Figure 4(a)). The result is that the check itself has relatively little apparent cost. Array bounds checks,

alternatively, require an explicit test and branch, and so eliminating them has a noticeable impact on the code being executed.

Surprisingly, the combination of eliminating both kinds of checks together is significantly more effective than the sum of eliminating both individually. This is a consequence of both optimization and the way null pointer checks have been implemented through the hardware exception mechanism. An array element reference in Java needs to be guarded by both a null pointer check and an array bounds check on the array index ([17]). In the High Performance Compiler, the code generated for an array bounds check naturally consists of a load of the array size followed by a comparison of the size with the index in question. The array size field, however, is offset only a small distance in memory from the start of the array object; the hardware supports trapping on a range of low addresses, and so a dereference of the array size field is as effective as dereferencing the object itself at generating the required hardware trap if the original object is null. Subsequent optimizations easily recognize this situation and combine the two checks into the same actual code; the code output for an array load or store is thus often identical whether a null pointer check is performed or not (see Figure 11(a)).

The symmetric situation, eliminating array bounds checks without also eliminating null pointer checks, is also not as effective as one might expect. In order to remove the array bound check while leaving behind the implicit null pointer check, specific code to dereference the array object must be inserted in order to still trigger the hardware trap mechanism if the array object is null (e.g. code in Figure 11(a) is replaced by the code in Figure 11(b)). This means that the benefit of deleting the bounds check code is offset slightly by the code required to explicitly dereference the object as part of a null pointer check.

<pre> mov eax,[ebx+offset] (implicit null ptr check) cmp eax,edx} jge outofbounderror </pre>	<pre> test eax,[eax] (explicit null ptr check) </pre>
--	---

(a) Array Bounds Check with
Implicit Null Pointer Check.

(b) Null Pointer Check Inserted if
Array Bounds Checks Eliminated.

Fig. 11. HPCJ Check Instructions

It is interesting that anomalous results occur in the FFT runs as well as the LU run of KaffeVM. Here the benchmark runs without some or all runtime checks are actually slower than the versions with checks. Since we are only reducing the checking overhead, and never increasing it, it seems counterintuitive that performance would ever be less than the baseline for any of our runs. However, in certain key benchmark functions the code changes due to eliminating some but not all bounds checks seems to negatively impact instruction cache utiliza-

tion, and we find our code highly sensitive to the exact sequence of bytes being executed. For instance, if the benchmark is compiled so as to artificially ignore the array bounds attribute information for a specific function (and thus generate normal bounds checks regardless of whether attribute information tells us they are unnecessary), much of the performance degradation is eliminated. The interaction of optimizing compilers with optimizing hardware is clearly a complex issue with many tradeoffs, and we may not be able to benefit all programs equally.

5 Related Work

Work related to this paper falls into three categories: (1) other tools for optimizing class files; (2) related techniques for optimizing array bounds checks; and (3) other uses of class file attributes.

5.1 Class File Tools

The only other Java tool that we are aware of that performs significant optimizations on bytecode and produces new class files is Jax[24]. The main goal of Jax is application compression where, for example, unused methods and fields are removed, and the class hierarchy is compressed. Their system is not focused on low-level optimization and it does not handle attributes.

There are a number of Java tools that provide frameworks for manipulating bytecode: JTrek[14], Joie[5], Bit[16] and JavaClass[12]. These tools are constrained to manipulating Java bytecode in their original form, however. They do not provide convenient intermediate representations such as Baf, Jimple or Grimp for performing analyses or transformations, they do not allow the production of bytecode, nor do they handle attributes.

5.2 Array Bounds Checks

Array bounds check optimization has been performed for other languages for a long time. Value range analysis has been used to remove redundant tests, verify programs, or guide code generation [9]. Further, there are a number of algorithms that use data flow analysis to remove partial redundant bounds checks [8,15].

More recently, the Java language has been the focus of research. Array bounds check elimination has been implemented in JIT compilers [22,4]. Midkiff et. al. [18,19] proposed a Java **Array** package and loop versioning algorithm to overcome the bounds check overhead in Java scientific programs. An algorithm for general applications was presented in [2]. Compared with these intraprocedural algorithms, our algorithm can take advantage of field information and our analysis for finding rectangular arrays.

Field analysis is useful to other optimizations such as object inlining and escape analysis [7]. Knowing an array's shape can also help memory layout of array objects [3].

5.3 Using Attributes

To the best of our knowledge there has been relatively little work done in investigating the possible uses of class file attributes to improve performance. We are aware of only two research groups that have been investigating this topic and both are focused on performance of bytecode.

Hummel et. al. gave an initial study on using attributes to improve performance, where they showed, using hand-simulation, that performance could be improved using attributes [10]. More recently this group has presented a system built on `guavac` and `kaffe` [1]. Their modified `guavac` compiler translates from Java source code to their own intermediate representation, and then converts this intermediate representation to annotated bytecode, which is then executed using their modified `kaffe` JIT. They have concentrated on conveying register allocation information. This involves developing a *virtual register allocation* scheme where one assumes an infinite number of registers and then proceeds to statically minimize the number that are actually used.

The second group, Jones and Kamin, have also focused on register allocation [13]. Their approach monotypes each virtual register, which allows for efficient runtime verifiability of their attributes; attributes to deal with spills are also presented. Their experimental results also exhibit significant code speedups.

Compared to these groups, our work is more focused on providing a general purpose framework for producing attributed class files. The input to our system can be class files produced by any compiler, and we provide the necessary infrastructure to convert the class files to typed 3-address intermediate code and to perform analysis on this intermediate code. We also provide a simple mechanism for converting the resulting flow analysis information to class file attributes. In this paper we have demonstrated how this scheme can be applied to array bounds checks; however, it can easily be applied to other problems, including the register allocation problem.

6 Conclusions and Future Work

In this paper we have presented an approach to using class file attributes to speed up the execution of Java bytecode. Our approach has been designed and implemented as part of the Soot bytecode optimization framework, a system that takes as input any Java bytecode and produces optimized and attributed bytecode as output. In our system, the compiler writer develops the appropriate flow analysis for the Jimple typed 3-address representation, and then uses tags to attach attribute information to the appropriate hosts (statements, methods, classes or fields). If the tags are attached to statements (**Units**), the compiler writer can also specify a tag aggregator which combines all tags within a method. The Soot system applies this aggregator when producing the attributed class files. As part of our system we have produced an extended version of the Jasmin assembler which can now support attribute directives and produces attributed class files.

Our system is very easy to use, and we showed how to apply it to the problem of eliminating array bound checks. We provided an overview of our array bounds check analysis and we showed how the results of the analysis can be propagated to attributes. We also modified two virtual machines, the Kaffe JIT and the IBM HPCJ ahead-of-time compiler, to take advantage of the attributes, and we presented experimental results to show significant performance improvements due to the array bounds check attributes.

In our current approach our attributes are not necessarily verifiable. Given verifiable class files and a correct analysis, we guarantee to produce correct attributes. Thus, if our system is used as a front-end to an ahead-of-time compiler, there is no problem. However, if our attributed class files are transmitted from our system to an external VM via an insecure link, we need a safety-checking mechanism to ensure the attribute safety. Nacula's proof-carrying code [20] could be one solution to this problem.

Based on our work so far, we believe that the attributes supported by our system can be used for a wide variety of tasks as outlined in the introduction, and we plan to work on several of these in the near future. In particular, we wish to examine how escape analysis and side-effect information can be expressed as attributes, and we plan to examine how some of the profile-based attributes can be used. We also would like to see how attributes can be used with other virtual machine implementations.

Acknowledgements. The authors would like to thank IBM CAS for supporting Raja Vallée-Rai with a IBM CAS Fellowship. This work was also supported by NSERC and FCAR. We would also like to thank Patrick Lam and Richard Godard for their work in implementing the first versions of null pointer analysis, which was the foundation for our null pointer attributes.

References

1. Ana Azevedo, Joe Hummel, and Alex Nicolau. Java annotation-aware just-in-time (AJIT) compilation system. In *Proceedings of the ACM 1999 Conference on Java Grande*, pages 142–151, June 1999.
2. R. Bodik, R. Gupta, and V. Sarkar. ABCD: Eliminating Array Bounds Checks on Demand. In *Proceedings of PLDI '00*, pages 321–333, June 2000.
3. M. Cierniak and W. Li. Optimizing Java bytecodes. *Concurrency, Practice and Experience*, 9(6):427–444, 1997.
4. Michal Cierniak, Guei-Yuan Lueh, and James M. Stichnoth. Practicing JUDO: Java under Dynamic Optimizations. In *Proceedings of PLDI '00*, pages 13–26, June 2000.
5. Geoff A. Cohen, Jeffrey S. Chase, and David L. Kaminsky. Automatic program transformation with JOIE. In *Proceedings of the USENIX 1998 Annual Technical Conference*, pages 167–178, Berkeley, USA, June 15–19 1998. USENIX Association.
6. Etienne M. Gagnon, Laurie J. Hendren, and Guillaume Marceau. Efficient inference of static types for Java bytecode. In *Proceedings of SAS 2000*, volume 1824 of *LNCS*, pages 199–219, June 2000.

7. S. Ghemawat, K.H. Randall, and D.J. Scales. Field Analysis: Getting Useful and Low-Cost Interprocedural Information. In *Proceedings of PLDI '00*, pages 334–344, June 2000.
8. R. Gupta. Optimizing array bound checks using flow analysis. *ACM Letters on Programming Languages and Systems*, 2(1-4):135–150, 1993.
9. W. Harrison. Compiler analysis of the value ranges of variables. *IEEE Transactions on Software Engineering*, 3(3):243–250, 1977.
10. Joseph Hummel, Ana Azevedo, David Kolson, and Alexandru Nicolau. Annotating the Java bytecodes in support of optimization. *Concurrency: Practice and Experience*, 9(11):1003–1016, November 1997.
11. The Jasmin Bytecode Assembler. <http://mrl.nyu.edu/meyer/jvm/jasmin.html>.
12. JavaClass. <http://www.inf.fu-berlin.de/~dahm/JavaClass/>.
13. Joel Jones and Samuel Kamin. Annotating Java class files with virtual registers for performance. *Concurrency: Practice and Experience*, 12(6):389–406, 2000.
14. Compaq-JTrek. <http://www.digital.com/java/download/jtrek>.
15. Priyadarshan Kolte and Michael Wolfe. Elimination of redundant array subscript range checks. In *Proceedings of PLDI '95*, pages 270–278, 1995.
16. Han Bok Lee and Benjamin G. Zorn. A tool for instrumenting Java bytecodes. In *The USENIX Symposium on Internet Technologies and Systems*, pages 73–82, 1997.
17. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999.
18. S. Midkiff, J. Moreira, and M. Snir. Optimizing bounds checking in Java programs. *IBM Systems Journal*, 37(3):409–453, August 1998.
19. J.E. Moreira, S.P. Midkiff, and M. Gupta. A Standard Java Array Package for Technical Computing. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, TX, March 1999.
20. G. Necula. Proof-carrying code. In *Proceedings of POPL '97*, pages 106–119, January 1997.
21. Soot - a Java Optimization Framework. <http://www.sable.mcgill.ca/soot/>.
22. T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java Just-in-Time Compiler. *IBM Systems Journal*, 39(1):175–193, 2000.
23. Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, and Étienne Gagnon. Practical virtual method call resolution for Java. In *Proceedings OOPSLA 2000*, pages 264–280, October 2000.
24. Frank Tip, Chris Laffra, Peter F. Sweeney, and David Streeter. Practical experience with an application extractor for Java. In *Proceedings OOPSLA '99*, pages 292–305, October 1999.
25. Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java Bytecode Optimization Framework. In *Proceedings of CASCON '99*, pages 125–135, 1999.
26. Raja Vallée-Rai, Etienne Gagnon, Laurie Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Proceedings of CC '00*, volume 1781 of *LNCS*, pages 18–34, March 2000.

SmartTools: A Generator of Interactive Environments Tools^{*}

Isabelle Attali, Carine Courbis, Pascal Degenne, Alexandre Fau, Didier Parigot, and Claude Pasquier

INRIA-Sophia – 2004 Route des Lucioles BP 93 06902 Sophia Antipolis Cedex
`{First.Last}@sophia.inria.fr`

Abstract. SmartTools is a development environment generator that provides a structure editor and semantic tools as main features. The well-known visitor pattern technique is commonly used for designing semantic analysis, it has been automated and extended. SmartTools is easy to use thanks to its graphical user interface designed with the Java Swing APIs. It is built with an open architecture convenient for a partial or total integration of SmartTools in other environments. It makes the addition of new software components in SmartTools easy. As a result of the modular architecture, we built a distributed instance of SmartTools which required minimal effort. Being open to the XML technologies offers all the features of SmartTools to any language defined with those technologies. But most of all, with its open architecture, SmartTools takes advantage of all the developments made around those technologies, like DOM, through the XML APIs. The fast development of SmartTools (which is a young project, one year old) validates our choices of being open and generic. The main goal of this tool is to provide help and support for designing software development environments for programming languages as well as application languages defined with XML technologies.

Keywords. Program transformation, Software development, Interactive Environment.

1 Introduction

Producing high-quality software has become a major concern in industry. There is a long history of research about providing help and support during the development process [6,7,9,11,14,15,16,21]. It is imperative that the research community creates technologies to enhance the quality of software development and increase the developers productivity. Those goals are addressed by the SmartTools framework and research. It is composed of a set of generic and interactive software components organized in a modular architecture.

^{*} This project is supported in part by Microsoft Research and Dyade, the Bull-Inria Research Joint Venture.

The reason for building a generic integrated environment, rather a collection of independent specific tools, is that integration enables the sharing of common services. It is also important, for each new language created, such as domain specific languages to propose and generate an interactive and uniform environment. The quality of these interactive environments must be as close as possible to those proposed by industrial distributors like the Visual Studio environment of Microsoft. Finally, the architecture of this system must be conceived with the aim of facilitating the integration of research tools and widely used tools (e.g. standardised Application Programming Interfaces, APIs).

These above requirements have justified our choice of Java and XML technologies. The SmartTools system is completely written with the Java programming language and strongly uses XML technologies. The SmartTools framework is a natural successor of the Centaur system [9] and uses the same basic concepts, as the abstract syntax tree (AST) specification or formalism.

From an AST specification describing a given language L, SmartTools automatically generates a structure editor dedicated to this language. So the end-user (developer) can edit any L file thanks to this uniform structure editor with generic visualisation tools (see Figure 1). For example, we can have many views of a document: one for the text representation, one for the structure editing menu and another for graphical representation.

With this basic environment, the designer of a given language is able to easily define and implement using the generic tools, a set of specific applications for his language. As we are using the Java Swing API, the graphical user interface has a great quality and is easily configurable.

In the following, we will only focus on the important aspects of SmartTools framework: the modular and extensible architecture, powerful and simple semantics tools and the utilisation of XML technologies. The system is open to other developments and uses standard mechanisms as XML for data exchange.

2 The Architecture of the SmartTools Framework

SmartTools is made of several independent software components that communicate with each other through an asynchronous messaging system. One component, the message controller, is in charge of controlling the flow of messages and delivering them to their destinations. All other components have to register on the message controller to let it know what types of messages they want to receive. Then they will receive only those types of messages and never be disturbed by the others. Information carried in messages is serialised in XML format. The design of this messaging system has proven to be simple, efficient, and easy to maintain.

Thanks to that component-based architecture, we designed a document/views relationship that gives us the possibility to have several different views showing one document each with a different presentation.

If for some specific need one wants to design a new type of view, it is not necessary for the designer of this new view to deal with all the program APIs.

It can be done by extending a default view component and adding some specific code to describe what happens when that view receives messages notifying any modification of the document. This design makes SmartTools seamlessly extensible for any kind of specific purpose.

It was not difficult using such a component-based architecture to take advantage of other tools developed at INRIA such as ProActive [4,11] to make a distributed version of SmartTools. Another benefit is that SmartTools components can be used by other applications without the need of the whole system. We made some successful experiences of SmartTools components integration in third parties environment [17,12,8].

We also easily connected new components like XML parsers (Sax, Xerces) and some graphic components like a graph server based on the Koala Graphic toolbox [3].

3 Semantic Analysis within the SmartTools Framework

For any tree manipulation, it is important to have a simple and powerful programming technique. This technique may neither require a high level of expertise nor an advanced knowledge about the tool.

With Java, an Object Oriented language, it is natural to use a well-known programming technic: the Visitor Design Pattern [13,18]. The advantage of this technique is the ability to reuse code and to specify dynamic semantic as an evaluator (see Figure 1). From the specification of an abstract syntax, it is possible to automatically generate Java source code, like a default depth-first traversal for example.

We have introduced a generic visitor concept to factorize identical behaviours applying to the nodes of a tree. With this technique we have defined one visitor only to graphically display the tree, for any language. We also use a Java Multi-Methods implementation [12] to fill many deficiencies [18,19] due to the visitor implementation based on the Java reflect APIs. Additionally with this technique, we gain lisibility without losing efficiency.

We use DOM (level 2) as Tree API where each node has the same type. In SmartTools we extend this behaviour by typing (sub-classing) each node according to a given formalism (AST). Thus, we can use the Visitor Pattern technique on DOM Tree. Many existing applications use visitor patterns (Generic Java [10], Dynamic Java [2], ...) and connecting them in SmartTools is very easy.

4 Using XML Technologies

As XML will be more and more used as a communication protocol between applications, we wanted to be able to handle any XML document in SmartTools [20]. Any XML document importing a DTD (Document Type Definition) has a typed structure. That DTD describes the nodes and their types, that is very similar to our AST formalism.

In order to obtain this result, we have specified and implemented a tool which converts a DTD formalism into an AST equivalent formalism. With this conversion, we automatically offer a structure editing environment for all languages defined with XML in the SmartTools framework. It is important to note that XML documents produced by SmartTools are well-formed.

So far, all the features of a DTD are not properly considered, by instance the importation of other DTDs (and namespaces), but that should be fixed in the near future. We are also studying XML schemas and RDF (Resource Description Framework) schemas, the successors of DTD.

Thus any application that respects the implementation of the APIs, can be XML-compliant. All the manipulated trees in SmartTools are Java DOM Trees to ease the integration with other tools and to have a very open data structure.

We offer a tool to automatically generate parsers. This tool can be useful for a designer to define a user-friendly concrete syntax for his language. But, extra data are required in the definition of the language.

We have also integrated the XSL (XML Style-sheet Language) specifications that describe the layout of a document as well as the XSLT (XSL Transformation).

5 Conclusion

SmartTools offers a quality development environment platform for research tools and benefits from large fields of applications thanks to XML technologies. The rapid evolution of SmartTools confirms our choices in term of modular architecture which facilitates the integration of other Java developments. In particular the choice of Java makes it possible to obtain a great quality graphical user interface with low development effort. Moreover the XML components, thanks to an open architecture, offers low cost advantages to SmartTools and broader application fields. We already have some examples of successful and easy integration of research tools [18,12], and technology transfer in industrial environment [5]. In both cases, the great quality of interactive environment, was the determinant element.

Acknowledgements. We have much benefited from discussions with Colas Nahaboo, Thierry Kormann and Stéphane Hillion on the topic of XML technologies. We would also like to thank Gilles Roussel, Etienne Duris and Rémy Forax for their helpful comments of their Java Multi-Methods implementation.

References

1. <http://marcel.uni-mb.si/lisa/>.
2. <http://www-sop.inria.fr/koala/djava/index.html>.
3. <http://www-sop.inria.fr/koala/koala.html>.
4. <http://www-sop.inria.fr/sloop/javall/index.html>.

5. <http://www.cp8.bull.net/odyssey/javaa.htm>.
6. Lex Augustejn. The elegant compiler generation system. In Pierre Deransart and Martin Jourdan, editors, *Attribute Grammars and their Applications (WAGA)*, volume 461 of *Lecture Notes in Computer Science*, pages 238–254. Springer-Verlag, New York–Heidelberg–Berlin, September 1990. Paris.
7. Don Batory, Bernie Lofaso, and Smaragdakis. JTS: A tool suite for building genovoca generators. In *5th International Conference in Software Reuse*, June 1998.
8. Frédéric Besson, Thomas Jensen, and Jean-Pierre Talpin. Polyhedral analysis for synchronous languages. In Agostino Cortesi and Gilberto Filé, editors, *Static Analysis*, volume 1694 of *Lecture Notes in Computer Science*, pages 51–68. Springer, 1999.
9. Patrick Borras, Dominique Clément, Thierry Despeyroux, Janet Incerpi, Gilles Kahn, Bernard Lang, and Valérie Pascual. CENTAUR: the system. *SIGSOFT Software Eng. Notes*, 13(5):14–24, November 1988.
10. Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the java programming language. In *Proc. OPPLA'98*, October 1998.
11. D. Caromel, W. Klauser, and J. Vayssiere. Towards seamless computing and meta-computing in java. In Geoffrey C. Fox, editor, *Concurrency Practice and Experience*, volume 10 of *Wiley and Sons, Ltd*, pages 1043–1061, September 1998.
12. Rémi Forax, Etienne Duris, and Gilles Roussel. Java multi-method framework. In *International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'00)*, November 2000.
13. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, 1995.
14. Martin Jourdan, Didier Parigot, Catherine Julié, Olivier Durin, and Carole Le Bellec. Design, implementation and evaluation of the FNC-2 attribute grammar system. In *Conf. on Programming Languages Design and Implementation*, pages 209–222, White Plains, NY, June 1990. Published as *ACM SIGPLAN Notices*, 25(6).
15. Uwe Kastens, Peter Pfahler, and Matthias Jung. The eli system. In Kai Koskimies, editor, *Compiler Construction CC'98*, volume 1383 of *Lect. Notes in Comp. Sci.*, portugal, April 1998. Springer-Verlag. tool demonstration.
16. Paul Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering Methodology*, 2(2):176–201, 1993.
17. Marjan Mernik, Nikolaj Korbar, and Viljem Zumer. LISA: A tool for automatic language implementation. *ACM SIGPLAN Notices*, 30(4):71–79, April 1995.
18. Jens Palsberg and C. Barry Jay. The essence of the visitor pattern. In *COMP-SAC'98, 22nd Annual International Computer Software and Applications Conference*, Vienna, Austria, August 1998.
19. Jens Palsberg, Boaz Patt-Shamir, and Karl Lieberherr. A new approach to compiling adaptive programs. In Hanne Riis Nielson, editor, *European Symposium on Programming*, pages 280–295, Linköping, Sweden, 1996. Springer Verlag.
20. Claude Pasquier and Laurent Théry. A distributed editing environment for xml documents. In *First ECOOP Workshop on XML and Object Technology (XOT2000)*, June 2000.
21. Thomas Reps and Tim Teitelbaum. The synthesizer generator. In *ACM SIGSOFT/SIGPLAN Symp. on Practical Software Development Environments*, pages 42–48. ACM press, Pittsburgh, PA, April 1984. Joint issue with *Software Eng. Notes* 9, 3. Published as *ACM SIGPLAN Notices*, volume 19, number 5.

Annex

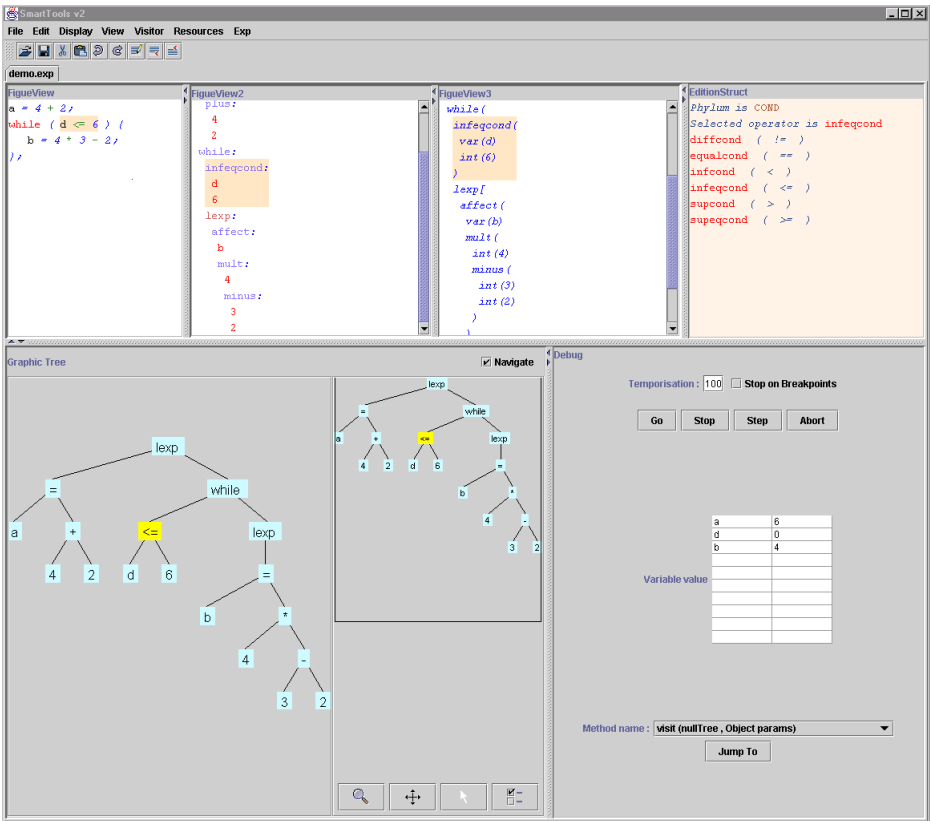


Fig. 1. SmartTools

Visual Patterns in the VLEli System

Matthias T. Jung, Uwe Kastens, Christian Schindler, and Carsten Schmidt

University of Paderborn, Fürstenallee 11, 33102 Paderborn, Germany,
 {mjung, uwe, tolkien, cschmidt}@upb.de

1 Characterization of the Toolset

Visual languages have an important role in modelling systems, in specification of software, and in specific application domains. A processor for a visual language consists of a graphical frontend attached to phases that analyse and transform the visual programs. Hence, the construction of a visual language processor requires a wide range of conceptual and technical knowledge: from issues of visual design and graphical implementation to aspects of analysis and transformation for languages in general. We present a powerful toolset that incorporates such knowledge up to a high specification level. Visual editors are generated by identifying certain patterns in the language structure and selecting a visual representation from a set of precoinced solutions. Visual programs are represented by attributed abstract trees. Hence, further phases of processing the visual programs can be generated by state-of-the-art tools for language implementation. We demonstrate that ambitious visual languages can be implemented with reasonable small effort and with rather limited technical knowledge. The approach is suitable for a large variety of visual language styles.

The main concepts of our approach are described by the three layers shown in Fig. 1. The tool VLEli generates visual structure editors from specifications. It is built on top of tools for graphical support (Tcl/Tk and Parcon) and for language implementation in general (Eli). The topmost layer contains variants of visual patterns, each of which encapsulates the implementation of visual language elements in terms of specifications for VLEli.

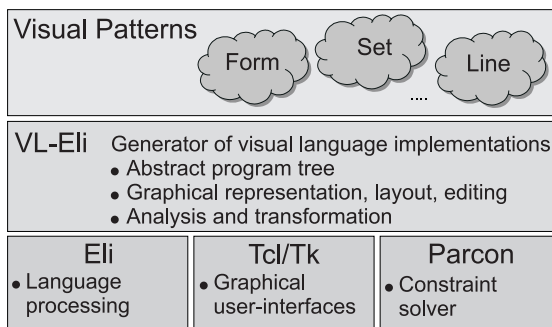


Fig. 1. Multi-Layered Specification

The central tool VLEli is conceptually based on attribute grammars: It generates visual editors, which operate on abstract program trees as their central data structures. They perform computations during tree walks as specified for the particular language. The computations create and modify the graphical representation of the program via the interface to the graphical support tools in the bottom layer. The abstract program tree and the attribute grammar method also enable the direct connection to many tools in the Eli system which solve a wide range of general language implementation tasks.

The topmost layer of our approach contains a set of precoinced descriptions of various graphical language elements in terms of the underlying layers. They are used to compose a visual language implementation without writing lower level specifications explicitly. Each such description is considered to be a variant of an abstraction that we call a *visual pattern*. It comprises the common properties of visual language elements with respect to their abstract structure, the interaction operations needed for them, and their visual concept. For example the **List** pattern represents an ordered sequence, allows element insertion and deletion, and visualizes the structure by drawing the elements side by side in a row. A language designer instantiates such a pattern variant and associates its components with certain constructs of the tree grammar.

2 Tool Demonstration

The demonstration of our tool set addresses three topics

1. Method: Visual patterns used in attribute grammar specifications.
2. Generated Products: Properties of the structure editors.
3. Variety of visual language styles.

2.1 Method

A visual pattern represents an abstract concept, like the visualization of a **set** and its elements. The notion of visual patterns provides also a means of reusable implementation specification: For each visual pattern concrete variants are described in terms of composable specifications for VLEli: Such a pattern variant encapsulates operations that are needed for a visual structure editor to implement a certain graphical representation of the structural abstraction. That are operations, which

- draw graphical components, e.g. the oval of a set,
- layout components of the structure, e.g. the elements of a set within its oval,
- provide facilities for user interactions, e.g. insert and delete elements.

These operations are expressed in terms of attribute grammar computations and composed to computational roles. The language designer selects such roles and associates them to the abstract syntax symbols. Fig. 2 gives an idea of various applications of patterns in five visual languages.

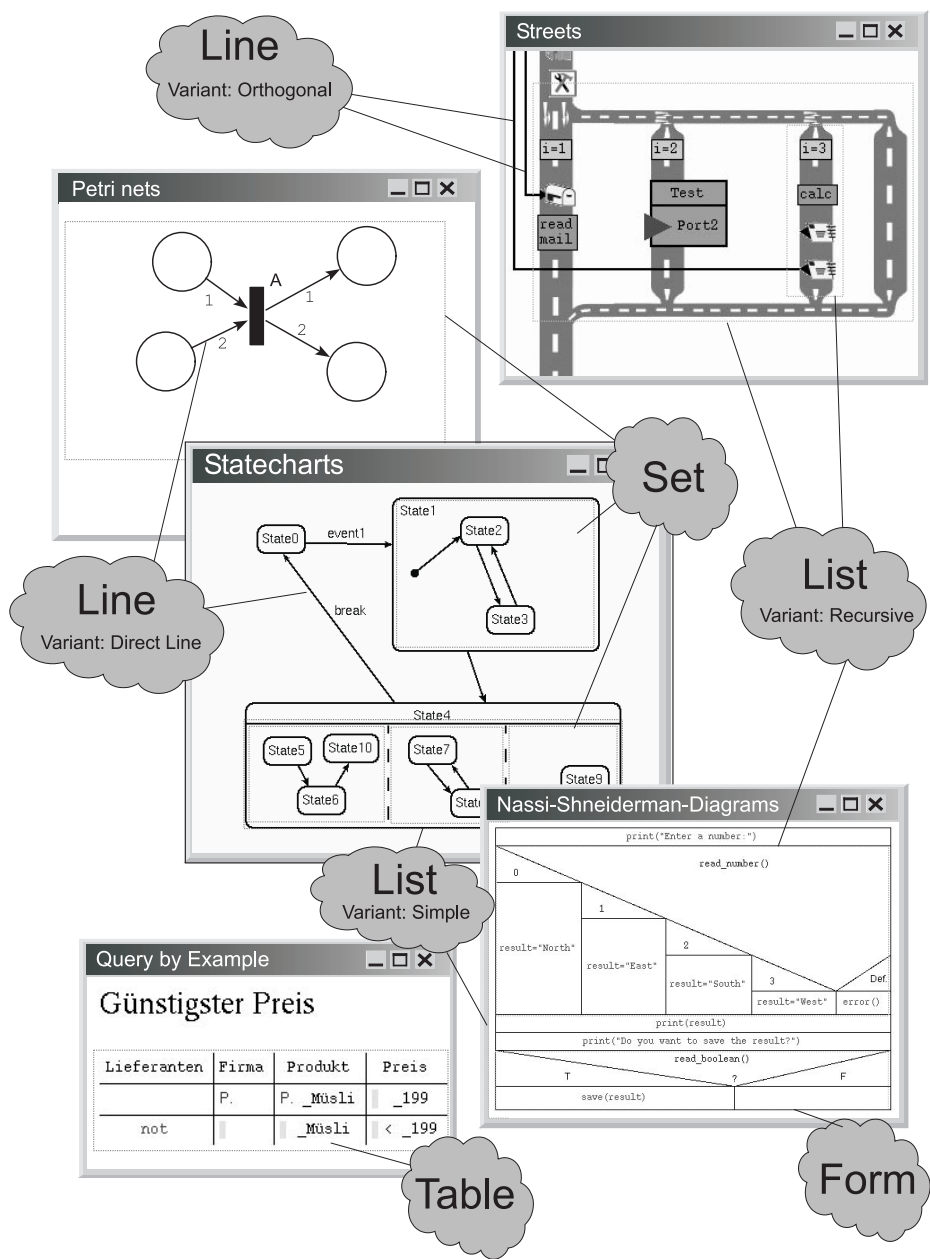


Fig. 2. Visual Pattern Applications in Visual Languages

2.2 Generated Products

The generated language processor consists of a graphical structure editor which builds an abstract program tree, and of analysis and translation phases which operate on that tree. The demonstration shows that the generated editors provide interactive graphical operations of high quality. The editors provide direct manipulation: The user can drag new language elements from toolbars and drop them where appropriate. The elements can be selected, and moved or deleted as required. Editing is assisted by highlighting the nearest possible location for moved language elements. Furthermore, all pattern variants provide specialized layout concepts. Thus a compact layout for the Nassi-Shneiderman-Diagrams is achieved as easily as the free non-overlapping layout of the statechart diagrams. For the latter, a constraint-solver is applied which adjusts the sizes and positions of language elements as necessary.

The implementation of these facilities comes almost for free by using variants of visual patterns.

2.3 Variety of Visual Language Styles

We checked the usability of our approach by implementing five visual languages: UML statechart diagrams and Petri-nets are examples of languages, that mainly apply the set-pattern and require a constraint-based layout to implement non-overlapping elements. Nassi-Shneiderman diagrams and Query-by-example are visual languages that use hierarchical structures visualized using automatic layout techniques. Query-by-example needs complex editing operations, e.g. to instantiate a database table. Streets is a more complex visual language for modeling parallel programs. Its implementation comprises a visual structure editor and phases for analysis and code-generation, which are generated from specifications for the Eli system.

The languages impose diverse requirements for editing and visualization features. They are implemented completely by visual patterns, in particular by variants of **Form**, **List**, **FormList**, **Table**, **Set** and **Line**. In the tool demonstration a selection of these generated products is shown to demonstrate the variety of visual language style that is addressed by our tool approach.

The ASF+SDF Meta-environment: A Component-Based Language Development Environment

M.G.J. van den Brand¹, A. van Deursen¹, J. Heering¹, H.A. de Jong¹, M. de Jonge¹, T. Kuipers¹, P. Klint¹, L. Moonen¹, P.A. Olivier¹, J. Scheerder², J.J. Vinju¹, E. Visser³, and J. Visser¹

¹ Centrum voor Wiskunde en Informatica (CWI), Kruislaan 413, 1098 SJ
Amsterdam, The Netherlands

² Faculty of Philosophy, Utrecht University, Heidelberglaan 8, 3584 CS Utrecht, The
Netherlands

³ Faculty of Mathematics and Computer Science, Utrecht University, Padualaan 14,
2584 CH Utrecht, The Netherlands

Abstract. The ASF+SDF Meta-environment is an interactive development environment for the automatic generation of interactive systems for constructing language definitions and generating tools for them. Over the years, this system has been used in a variety of academic and commercial projects ranging from formal program manipulation to conversion of COBOL systems. Since the existing implementation of the Meta-environment started exhibiting more and more characteristics of a legacy system, we decided to build a completely new, component-based, version. We demonstrate this new system and stress its open architecture.

1 Introduction

The ASF+SDF Meta-environment [12] is an interactive development environment for the automatic generation of interactive systems for constructing language definitions and generating tools for them. A language definition typically includes such features as syntax, prettyprinting, typechecking, and execution of programs in the target language. The ASF+SDF Meta-environment can help in the following cases:

- You have to write a formal specification for some problem and you need interactive support for this.
- You are developing your own (application) language and want to create an interactive environment for it.
- You have programs in some existing programming language and you want to analyze or transform them.

The ASF+SDF formalism [1] [10] allows the definition of syntactic as well as semantic aspects. It can be used for the definition of languages (for programming, writing specifications, querying databases, text processing, or other

applications). In addition it can be used for the formal specification of a wide variety of problems. ASF+SDF provides:

- A general-purpose algebraic specification formalism based on (conditional) term rewriting.
- Modular structuring of specifications.
- Integrated definition of lexical, context-free, and abstract syntax.
- User-defined syntax, allowing you to write specifications using your own notation.
- Complete integration of the definition of syntax, and semantics.
- Traversal functions (for writing very concise program transformations), memo functions (for caching repeated computations), and more.

The ASF+SDF Meta-environment offers:

- Syntax-directed editing of ASF+SDF specifications.
- Incremental compilation and testing of specifications.
- Compilation of ASF+SDF specifications into dedicated interactive stand-alone environments containing various tools such as a parser, prettyprinter, syntax-directed editor, debugger, and interpreter or compiler.
- User-defined extensions of the default user-interface.

The design goals of the new implementation to be demonstrated include: openness, reuse, extensibility, and in particular the possibility to generate complete stand-alone environments for user-defined languages.

2 Technological Background

ToolBus. A hallmark of legacy systems in general and the old ASF+SDF Meta-environment in particular is the entangling of control flow and actual computation. To separate coordination from computation we use the ToolBus coordination architecture [2], a programmable software bus based on process algebra. Coordination is expressed by a formal description of the cooperation protocol between components while computation is expressed in components that may be written in any language. We thus obtain interoperability of heterogeneous components in a (possibly) distributed system.

ATerms. Coordination protocol and components have to share data. We use ATerms [4] for this purpose. These are trees with optional annotations on each node. The annotations are used to store tool-specific information like text coordinates or color attributes. The implementation of ATerms has two essential properties: terms are stored using maximal subterm sharing (reducing memory requirements and making deep equality tests very efficient) and they can be exchanged using a very dense binary encoding that preserves sharing. As a result very large terms (with over 1,000,000 nodes) can be processed.

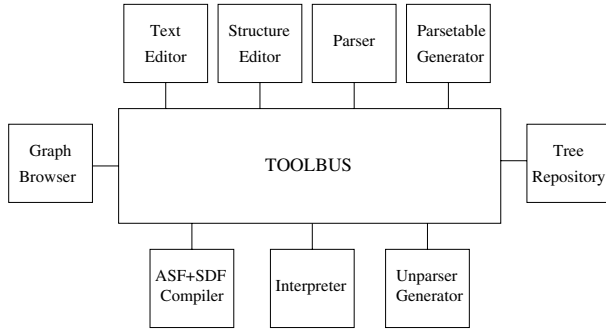


Fig. 1. Architecture of the ASF+SDF Meta-environment

SGLR. In our language-centric approach the parser is an essential tool. We use scannerless, generalized-LR parsing [13]. In this way we can parse arbitrary context-free grammars, an essential property when combining and parsing large grammars for (dialects of) real-world languages.

Term rewriting. ASF+SDF specifications are executed as (conditional) rewrite rules. Both interpretation and compilation (using the ASF2C compiler [5]) of these rewrite rules are supported. The compiler generates very efficient C code that implements pattern matching and term traversal. The generated code uses ATerms as its main data representation, and ensures a minimal use of memory during normalization of terms.

3 Architecture

The architecture of the ASF+SDF Meta-environment is shown in Figure 1. It consists of a ToolBus that interconnects the following components:

- **User interface:** the top level user-interface of the system. It consists primarily of a graph browser for the import graph of the current specification.
- **Text Editor:** a customized version of XEmacs for text editing.
- **Structure Editor:** a syntax-directed editor that closely cooperates with the Text Editor.
- **Parser:** scannerless, generalized-LR parser (SGLR) that is parametrized with a parse table.
- **Parsetable generator:** takes an SDF syntax definition as input and generates a parse table for SGLR.
- **Tree Repository:** stores all terms corresponding to specification modules, parse tables, user-defined terms, etc.
- **Compiler:** the ASF2C compiler.
- **Interpreter:** executes specifications by direct interpretation.
- **Unparser generator:** generates prettyprinters.

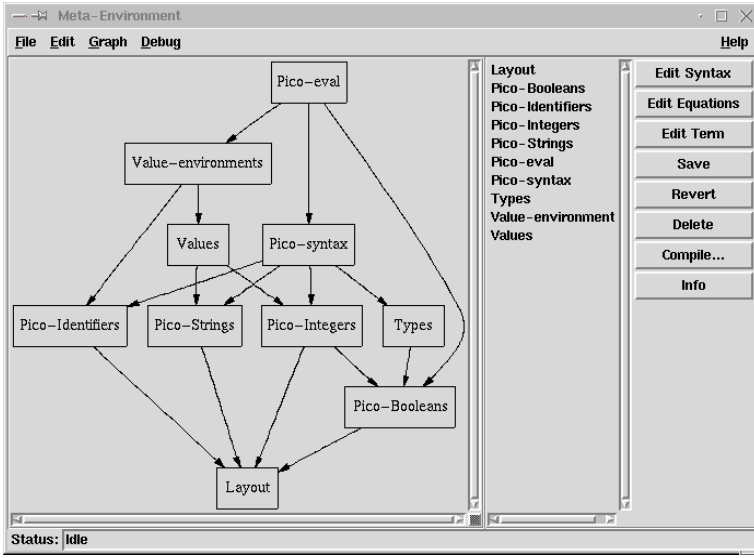


Fig. 2. The main user-interface of the Meta-environment is a module browser that provides a graphical and a textual view of the modules in a specification. A number of operations can be initiated for each module. Here it is shown with the modules from a small specification of a typechecker for the toy language Pico.

4 Applications of ASF+SDF and the Meta-environment

There are a number of academic and industrial projects that use either ASF+SDF directly or components of the Meta-environment in one way or another. The applications of ASF+SDF can be split into three groups:

1. In the field of language prototyping ASF+SDF has been used to describe the syntax and semantics of *domain specific languages*, e.g., the language Risla for describing financial products [3]. As another example, the syntax of the algebraic specification language CASL has been prototyped using ASF+SDF [7]. BOX [9, 11] is a small domain specific language developed for prettyprinting within the Meta-environment.
2. In the field of *reverse engineering and system renovation*, ASF+SDF is used to analyze and transform COBOL legacy code [8].
3. As an algebraic specification formalism for specifying *language processing tools*. In fact, a number of components of the Meta-environment itself have been specified using ASF+SDF:
 - the ASF2C compiler,
 - the unparser generator, and
 - parts of the parsetable generator.

For other components, such as the ToolBus and the syntax-directed editor, an ASF+SDF specification was made for prototyping use only. That specification formed the basis for an optimized, handcrafted implementation.

Components of the Meta-environment are used as stand-alone tools in a variety of applications. Examples are the Stratego compiler [14], the Risl compiler, the Elan environment [6], and a commercial tool for designing and implementing information systems.

5 Demonstration

We will show a number of applications of the Meta-environment ranging from a simple typechecking problem (Figure 2) to syntax-directed editing and transformation of COBOL systems.

6 Obtaining the ASF+SDF Meta-environment

The ASF+SDF Meta-environment can be downloaded from:

<http://www.cwi.nl/projects/MetaEnv/completa/>

Individual components, such as the ATerm library, ToolBus, parser generator, and parser (SGLR) can be obtained from: <http://www.cwi.nl/projects/MetaEnv/>.

All components of the ASF+SDF Meta-environment are available as open source software.

References

1. J.A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification*. ACM Press/Addison-Wesley, 1989.
2. J.A. Bergstra and P. Klint. The discrete time ToolBus – a software coordination architecture. *Science of Computer Programming*, 31(2-3):205–229, July 1998.
3. M.G.J. van den Brand, A. van Deursen, P. Klint, S. Klusener, and E.A. van den Meulen. Industrial applications of ASF+SDF. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology (AMAST '96)*, volume 1101 of *LNCS*. Springer-Verlag, 1996.
4. M.G.J. van den Brand, H.A. de Jong, P. Klint, and P. Olivier. Efficient Annotated Terms. *Software, Practice & Experience*, 30:259–291, 2000.
5. M.G.J. van den Brand, P. Klint, and P. A. Olivier. Compilation and memory management for ASF+SDF. In S. Jähnichen, editor, *Compiler Construction (CC '99)*, volume 1575 of *Lecture Notes in Computer Science*, pages 198–213. Springer-Verlag, 1999.
6. M.G.J. van den Brand and C. Ringeissen. ASF+SDF parsing tools applied to ELAN. In *Third International Workshop on Rewriting Logic and Applications*, ENTCS, 2000.
7. M.G.J. van den Brand and J. Scheerder. Development of Parsing Tools for CASL using Generic Language Technology. In D. Bert, C. Choppy, and P. Mosses, editors, *Workshop on Algebraic Development Techniques (WADT'99)*, volume 1827 of *LNCS*. Springer-Verlag, 2000.

8. M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. *Science of Computer Programming*, 36:209–266, 2000.
9. M.G.J. van den Brand and E. Visser. Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology*, 5:1–41, 1996.
10. A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific, 1996.
11. M. de Jonge. A pretty-printer for every occasion. In I. Ferguson, J. Gray, and L. Scott, editors, *Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools (CoSET2000)*. University of Wollongong, Australia, 2000.
12. P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2:176–201, 1993.
13. E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997.
14. E. Visser, Z. Benaissa, and A. Tolmach. Building Program Optimizers with Rewriting Strategies. In *International Conference on Functional Programming (ICFP'98)*, pages 13–26, 1998.

Author Index

- Amaral, José Nelson 289
Araujo, Guido 274
Attali, Isabelle 355
Aycock, John 229

van den Brand, M.G.J. 365

Courbis, Carine 355

Debray, Saumya 319
Degenne, Pascal 355
Dehnert, James C. 289
van Deursen, A. 365
Doshi, Gautam 165
Douillet, Alban 289

van Engelen, Robert A. 118

Farré, Jacques 244
Fau, Alexandre 355
Fortes Gálvez, José 244
Franke, Björn 69

Gao, Guang R. 289
Graham, Susan L. 102
Gregg, David 200
Gregor, Douglas 86

Heering, J. 365
Hendren, Laurie 334
Horspool, Nigel 229

Jain, Suneel 289
de Jong, H.A. 365
de Jonge, M. 365
Jung, Matthias T. 361

Kandemir, Mahmut Taylan 259
Kastens, Uwe 361
Kim, Jihong 182
Klint, P. 365
Kuipers, T. 365

Lacey, David 52
Lehrman Madsen, Ole 1
Liu, Shin-Ming 86

Malik, Sharad 274

Martena, Vincenzo 3
Mehofer, Eduard 37
Moon, Soo-Mook 182
Moonen, L. 365
de Moor, Oege 52
Musser, David 86
Muthukumar, Kalyan 165

O'Boyle, Michael 69
Olivier, P.A. 365
Ottoni, Guilherme 274

Parigot, Didier 355
Pasquier, Claude 355
Pominville, Patrice 334

Qian, Feng 334

Rajagopalan, Subramanian 274
Rigo, Sandro 274
Rinard, Martin 150
Rinetzky, Noam 133
Rountev, Atanas 20
Rugina, Radu 150
Ryder, Barbara G. 20

Sagiv, Mooly 133
San Pietro, Pierluigi 3
Scheerder, J. 365
Schindler, Christian 361
Schmidt, Carsten 361
Scholz, Bernhard 37
Schupp, Sibylle 86
Siebert, Fridtjof 304
Stoutchinin, Artour 289

Tice, Caroline 102
Touati, Sid Ahmed Ali 213

Vallée-Rai, Raja 334
Verbrugge, Clark 334
Vinju, J.J. 365
Visser, E. 365
Visser, J. 365

Watterson, Scott 319

Yun, Han-Saem 182